Nebulostore: an incentive-compatible, common storage layer for decentralized online social networks

Krzysztof Rzadca
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw, Poland
krzadca@mimuw.edu.pl

Gunnar Kreitz
KTH Royal Institute of Technology, Sweden
gkreitz@kth.se

Boleslaw Kulbabinski
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw, Poland
bkulbabinski@gmail.com

Sonja Buchegger
KTH Royal Institute of Technology, Sweden
buc@kth.se

Jadwiga Kanska
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw, Poland
jk277560@students.mimuw.edu.pl

ABSTRACT

Persistent and highly available storage is a fundamental building block of an Online Social Network (OSN). Existing approaches use either organizationally-centralized or decentralized methods. In organizationally-centralized systems, a single organization (Facebook, Google, etc.) keeps all the OSN data; such approaches are highly efficient, but are susceptible to data ownership and control issues, such as monitoring and censorship. Recent controversies over Facebook privacy settings, the European Union's data protection regulation proposal on the right to be forgotten, or the PRISM monitoring system show that users are starting to be aware of these problems. Decentralized systems using free, community-certified software might offer a more privacy-preserving alternative. However, in decentralized approaches it is notoriously difficult to achieve availability, consistency, and persistence.

In this paper we describe Nebulostore, a decentralized storage system that offers both high availability and consistency. The raw storage space is provided by integrating diverse storage resources, such as PCs, plug computers, and cloud shares. Nebulostore uses game-theoretic incentives for end users to keep their resources highly available. The heterogeneity of storage space is abstracted through two-level addressing and an API based on lists. The lists allow us to maintain consistency of concurrently-modified objects (such as a user's "wall").

We show that the proposed API is sufficient for supporting, in a privacy-preserving manner, common decentralized on-line social network (DOSN) use cases, such as user-to-user messaging, walls, or galleries. Our experiments demonstrate that the prototype implementation has adequate performance. Our implementation is released with an open-source license and available at nebulostore.org. We hope that Nebulostore becomes a standard storage module that can be used by prototype implementations of DOSNs and other large-scale, community-oriented applications.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed Systems*

General Terms

Distributed Online Social Networks, incentive-compatible

1. INTRODUCTION

Developing a distributed file system is a known difficult problem in terms of availability, consistency, and persistence on the technical side, as well as incentives for cooperation, dealing with free-riding and heterogeneous resources and demands on the economics and viability side. It is a relatively old area of research that aimed at replicating the functionality of systems such as regular, centralized file systems, backup, or file sharing, but in a distributed way. Most of the research activity in this area has slowed down. The era of social media and particularly of online social networks, however, has brought up privacy concerns over the (logically) centralized collection of data that can be mined, correlated, monitored, or censored, thus leading to the idea of decentralizing online social networks (DOSN) to take away the centralized threat of accidental or deliberate privacy breaches.

DOSN, however, also need reliable, available, consistent, and persistent storage. In addition to the challenges of a regular distributed file system, a DOSN has properties that pose different requirements to an underlying storage or file system. Unlike a backup system, content is shared with many other users, other users can both read some parts of the content and write to some other users' files, according to certain access control policies. Content can be retrieved frequently, by many users, and this retrieval is time sensitive if usability is to be taken into account. The files that need to be stored can be anywhere on the spectrum of static large media files to small status updates that change frequently, whereas file-sharing applications mainly have content that does not change. In DOSN, recent updates need to be prioritized. DOSN need storage that is lightweight and flexible enough to accommodate a range of application and communication protocols with different requirements concerning update frequency, user read/write access rights and frequency, availability of newer vs. older content, etc. for functions as diverse as wall postings, news and friend feeds, commenting, private messaging, media collections like photo albums, requiring both deliberate user actions and protocol-level actions by the software agents representing the users.

Moreover, information in DOSN can be of a very personal nature and needs to be protected in an untrusted and potentially adverserial environment where there is no control over another storage node's operating system yet the content needs to be accessible only for an authorized, but typically large number of users; there is no common organizational unit.

In the extreme case, DOSN is fully hosted and replicated on (other) end users' untrusted storage, in contrast to a central provider as is the case with current online social networks. One could also imagine a cloud-based system instead. While a full cloud system retains problems similar to the centralized provider case (privacy, dependency), a system that allows for integrating cloud storage is an option toward a privacy-preserving freemium service without ad-funding. In this service, a user can choose between paying for availability (cloud) or using it for free (peer-to-peer). We envision a clear separation between the developer of the service software and the infrastructure provider, so we cannot simply overcharge paying customers to subsidize cloud storage for the free riders. In this paper, we focus on the fully distributed case for an incentive-compatible storage for DOSN.

1.1 Our Contributions

The main contribution of this work is the design of an incentive-compatible storage system sophisticated enough for DOSN use cases, but possible to implement and with a prototype tested on a real distributed network. In particular, we propose an addressing scheme that makes a uniform addresses out of a very heterogeneous storage space (section 3.2). Our design further includes a new interface (API) between the DOSN and the storage system based on lists (section 4). While a storage system is a generic component, the API design was driven by a DOSN use case, and we also showcase parts of a DOSN system design using the API.

In addition to the design, we also provide an open source implementation of the storage system, available from `nebulostore.org`.

In the remainder of the paper, we present the different elements of the storage system (section 3), the interface to applications such as DOSN (section 4), security considerations (section 5), an example of a concrete DOSN making use of the storage system (section 6), and an experimental evaluation of the prototype (section 7).

2. RELATED WORK

In this paper, we provide the formulation of an interface for the storage layer that can be both supported by the storage layer and sophisticated enough for complex DOSN use cases. While there have been several proposals for DOSN, they have not demonstrated availability and consistency on a large scale. Our proposed storage layer and interface is available and its performance is evaluated in this paper; it can be reused by various DOSN approaches and thus allow researchers and developers to focus on DOSN functionality.

Nebulostore builds the storage space using game-theoretic incentives. In contrast, early distributed storage systems relied on altruism [23, 22, 1, 39, 11, 17, 9, 37, 12, 31]: for instance, in a classic DHT, an agent may or may not contribute its bandwidth and storage space; and the amount of contributed resources does not impact the quality of perceived service (e.g., the size of the data that the agent can store). Rational agents free-ride in such systems; indeed, in a 2005 study of the gnutella network [15] (an early file-sharing network with no protocol-level incentives for sharing), 85% of peers free-rode by not sharing any files.

In incentive-compatible storage systems, users are modelled as autonomous agents, and they seek to maximize their perceived profits (e.g., availability of their data) and to minimize their contribution (e.g., the amount of other users' data they store) [30, 2, 5, 29]. Recent implementations of incentive-compatible storage or the personal-cloud systems include e.g. `spacemonkey.com`, `symform.com`, or [6]. The primary goal of all these systems is backup and file sharing; thus they do not support higher-level DOSN use cases.

The approach taken with lists can be seen as an extension of the Alternate Data Stream feature of the NTFS file system used by Windows. In that file system, each file is in fact a collection of byte streams, with most files consisting only of a single stream. However, usage of alternate data streams remained low in part due to lack of tool support and expectations on a file system.

In relation to our case study of a DOSN running on top of Nebulostore, there is a rich literature of DOSNs. Many of these [3, 10, 16, 33, 32] follow a similar design pattern as in our case study: building a DOSN application on top of a storage layer, with confidentiality enforced by cryptographic means in the DOSN layer.

The degree of reliance in the integrity of storage varies between the approaches.

An alternative to DOSN is to build a more privacy-aware OSN by using organizationally-centralized storage and encryption (e.g. Facebook servers [20]). However, this infrastructure must be somehow payed for. Currently, cloud storage providers offer storage as a freemium service: while e.g. the first GB is free, 1 TB of cloud storage costs approximately \$1000 per year. Moreover, in this setting, the network becomes dependent on the storage provider which may, e.g., try to analyze traffic, impose terms of service that are not privacy friendly, or even disappear overnight.

There is also a class of system where the storage system is more directly integrated into the social network design, by enlisting friends to store or cache a users data [13, 34, 25, 36]. Such systems would not directly translate onto our storage, but one could imagine a system similar to Nebulostore fulfilling the role of the superpeers used in SuperNova [34].

None of the aforementioned DOSN approaches explicitly builds on lists, but we argue that most of the designs could be straightforwardly ported to make use of a list-based storage system. Even if one does not change the design to be list-based, one could port many of the approaches to run on a Nebulostore system where the list functionality is simply ignored and only a single list entry is used in each list, making the functionality close to other distributed storage systems.

3. OVERVIEW OF THE STORAGE SYSTEM

Our storage system organizes a uniformly-addressable storage space on infrastructure defined by pair-wise replication contracts [30]. In this section, we first describe the notion of pair-wise contracts and show how the contracts give incentive to users to contribute high-quality resources. This infrastructure, however, is highly heterogeneous: we next show how the addressing layer provides a uniform naming space over these heterogeneous resources. Then, we discuss additional features that must be provided by the storage layer for DOSN functionality. The presentation in this section abstracts from security requirements, which are discussed in section 5.

3.1 Incentive-compatible replicated storage space

In large-scale systems coordinating and sharing resources, only a minority of users is altruistic [15]; a majority behaves rationally and, if only possible, prefers to free-ride on the resources made available by others. The fundamental design principle of our storage system is to be incentive-compatible: to reward users contributing high-quality resources with better quality of service. This section sketches the intuition behind the bilateral contracts; see our earlier paper [30] for an analytical as well as an experimental analysis.

The basic element of Nebulostore is a *bilateral storage contract*: a contract between two agents to mutually store and serve a certain amount of each other's data for a certain time at a certain availability level. An example contract between agents i and j may allow i to upload e.g. 1 GB of data to a device controlled by j and then binds j to serve the data to other users for e.g. the next two weeks (so that if another agent k wants to see i's collection of photos, it can download it from j). The contract also specifies that the data should be available to download from j at least e.g. 90% of the time. The contract is bilateral: i must allow j to upload the same amount of data and then to serve it for the same period (but not necessarily at the same availability level). A party can terminate the contract earlier if the other party does not comply (for instance, by serving the data at much lower availability).

The notion of the contract is general and can be extended to other scenarios. For instance, a 1 GB data contract may automatically expire after, e.g., two weeks or 10 GB of used download bandwidth. Similarly, the service level agreement may specify either the average availability (e.g. at least 95% of the week the data must be available for others to download), or the average availability in certain time slots (e.g. 95% between 9am and 5pm GMT during week-days to match uptimes/downtimes similarly to [7, 19]). The time-slot model allows the system to use the spare capacity of existing resources (such as office PCs) almost without additional energy consumption. The contracts can even be asymmetric [27]: weakly-available agents may contribute more resources to replicate data with highly-available agents.

As no device is perfectly reliable, in order to increase availability, each agent replicates its data using multiple storage contracts with different agents. We model a device's non-availability period as its transient failure. A single available device is sufficient to download the data. Assuming that transient failures are independent [2, 5], and that the data is fully replicated, the probability that the data is unavailable is a product of the unavailabilities of individual contracts (the service level agreements). For instance, data replicated on three different agents with contracted availabilities of 90%, 95% and 99% is available with probability $0.9995 = 1 - 0.1 \cdot 0.05 \cdot 0.01$. However, other, more sophisticated replication strategies can be used: erasure coding [26, 28] or replication of different data on different subsets of contracts.

In our earlier work [30] we used a game theoretic model to analyze the distribution of contracts in a society of rational agents. The main conclusion is that the bilateral contracts give incentive to agents to be highly available: as each agent wants its data to be highly available, it seeks to have contracts with highly-available partners. Given that the agents' bandwidth and storage space is limited, an agent will choose highly-available replication partners. Thus, for instance, an agent with e.g. 90% average availability seeking a single contract will prefer an agent with 89% availability to an agent with 85% availability (of course, an agent with 99% availability would be even better, but this agent would normally have better candidates for storage partners than a 90%-availability agent). Consequently, the higher is the agent's availability, the higher is its data availability, thus the better quality of service the agent gets. Importantly, we do not need to use out-of-system incentives, such as monetary payments: using (or even mentioning) money discourages people from cooperating [38].

The bilateral contracts also allow our system to integrate the storage space provided by agents' devices (PCs, plug PC, smart routers, etc.) with cloud storage that some agents might prefer to contribute. A user has to pay for cloud resources, but in return she will be able to form storage contracts with other highly-available agents (who contribute either cloud storage or highly-available devices).

3.2 Two-Level Addressing

The storage space provided by the bilateral storage contracts is highly non-uniform. The data owner may have different replication strategies for different objects. Moreover, over time the availability of an agent changes, and thus the contracts may change. Finally, as a user may have a device that migrates between networks, its (network) address may change. The addressing layer abstracts from these issues by two directories: the object directory and the device directory.

The primary goal of the addressing layer is to provide a fast look-up: given the ID of the object, the addressing layer should quickly return a list of addresses of its current replicas. Moreover, the addressing layer should scale sub-linearly with the number of objects. The addressing layer is a public service: while all users benefit from using the addressing infrastructure, it is difficult to design incentives that reward users donating their resources for it. Consequently, we decided to use a super-peer-like approach: we keep the addressing layer on a small number of highly-available resources that are donated altruistically; but we also limit the amount of information stored and transmitted by these nodes.

The addressing layer is kept on a DHT maintained by highly-available nodes (highly-available devices or cloud shares donated to the network). The *device directory* resolves a device ID to the current network address of the Nebulostore running on that device (for instance, the IP address and the port). The *object directory* resolves the object ID to a set of device IDs that replicate the object (see section 5 for security considerations of this design).

A naive implementation of the object directory requires an entry for each object, which would result in high storage and bandwidth costs for the DHT nodes. To reduce the number of DHT entries, we adopt a special naming (addressing) of objects. An object ID is a tuple `(bucketID, objectName)`. The idea is that related objects (e.g.: profile information, pictures and videos for a single DOSN user) have the same `bucketID`, but different `objectNames`. The DHT stores an entry for each `bucketID`. The information stored in the entry resolves each `objectName` to a set of replicators (device IDs) storing the object.

More formally, the DHT entry is a list of tuples: `(th1,{d1-1,d1-2,...,d1-k1}),(th2,{d2-1,d2-2,...,d2-k2}),...,(thn,{dn-1,dn-2,...,dn-kn})`. The objects with `0<objectName≤th1` are replicated by devices with device IDs `{d1-1,d1-2,...,d1-k1}`; the objects with `th1<objectID≤th2` are replicated by devices with device IDs `{d2-1,d2-2,...,d2-k2}`, and so on. In order to get an object with an ID e.g. `(123, 456)`, a client connects to the DHT and gets the list of tuples, (e.g.: `(300, {abc, efg}), (600, {abc, xyz}), ...`). By comparing the `objectName` with subsequent threshold values (300, 600, ...), the client knows that the object is stored at devices `abc` and `xyz`. Using another DHT query to the device directory, the client then resolves one of the replicators (e.g. `abc`) to get its current network address; finally, the client connects to the replicator to download the object. In practice, because the client usually refers to multiple objects from the same bucket, the result of the device query will be probably found in the client's local cache. If the client can't connect to the replicator, it resolves and connects to the second replicator and so on (to reduce the latency, queries to the device directory may be done in parallel).

The two-level object IDs reduce the number of entries in the DHT from $\mathcal{O}$`(# objects)` to $\mathcal{O}$`(# buckets)`. As a DOSN stores the complete "persona" of a user in a single bucket, the number of DHT entries is significantly reduced. At the same time, the size of an entry is not large: it depends on the space of the underlying replication agreements, but we can roughly expect e.g. 1GB replication agreements, which result in up to 100 entries for a user storing 100GB of data. Finally, by keeping the range of `objectName` large (e.g.: 2^{32}), we may reserve the 8 most significant bits to encode the subsequent threshold values, which leaves 2^{24} possible objects in each bucket: a sufficient addressing space to be able to, e.g., balance the free space across the buckets. Larger `objectName` spaces (e.g.: 2^{128}) enable to probabilistically guarantee that it is hard to "guess" the object ID. Note that if a replicator fails or a contract is changed, the new replicator takes the old replicator's role.

Associating a DOSN-level "persona" with storage-level `bucket id` might result in privacy risks. However, privacy-preserving DOSN can implement their own measures to prevent traffic analysis or privacy breaches on the storage layer. This way inferences about user data from `bucket id`s can be mitigated.

3.3 Subscriptions

Several of the DOSN use cases (section 6) require an instance of the DOSN layer to be notified when an object changes. Nebulostore offers a subscription mechanism in which one Nebulostore instance observes the changes at one of the replicators of a target object (the pull model). We are currently working on a more complex mechanism that will reduce the burden on the replicators; however, this is out of the scope of this paper.

3.4 Versioning and propagation of updates

In Nebulostore, each object has a single agent who *owns* it. The owner is responsible for the replication space for storing the object; and is also able to arbitrary modify (or to delete) the object. (Nebulostore also supports appends to lists that can be done by other agents, see section 4). This section describes the protocol used by the owner to update an object and its replicas; parts of this protocol are used by the list append operation (section 4.2).

In principle, Nebulostore does not support concurrent arbitrary updates: as there is only a single user who may do such updates, concurrent updates should not be needed (concurrent appends to list objects, such as wall posts or comments can be done by other users and are treated in section 4). However, if concurrent updates are needed, an election protocol at an upper layer can be designed (if the set of replicators is stable).

The main problem that remains is to propagate updates to replicators that may be offline for long time periods. To this end, Nebulostore needs to keep track of the sequence of updates. This is done through a version ID which is a list $(k_1, k_2, \dots, k_n)$ of random numbers. On each update, the owner generates a new random number r. This number is appended to the end of the version ID list: $(k_1, k_2, \dots, k_n, r)$. The list version IDs enable the replicators to detect concurrent updates. Assume that one replicator stores an object with a version ID $K = (k_1, k_2, \dots, k_{n1})$, while the other has the same object with a version ID $L = (l_1, l_2, \dots, l_{n2})$. An update is not concurrent if and only if one list is a sublist of the other (i.e. for all $1 \leq i \leq \min(n_1, n_2)$, $l_i = k_i$). In case of conflict, the owner is notified and the versions are not merged (until the owner manually resolves the conflict). If the list grows too long, it can be pruned by deleting the parts all replicas have; in order to prune the list, all replicators perform voting to get the maximum common list and then a distributed commit to prune the agreed common list.

After generating the new version ID, the owner creates a transaction to actually modify the files following the 3-Phase Commit protocol (however, in our case the problem is simpler as a replicator can refuse a commit only if it is offline). The owner contacts a few replicators and uploads the new version of the object (the upload might be done by application-layer multicast protocols (such as [14]); the transaction is completed successfully (committed) when a pre-defined number of replicators acknowledged the transmission. Otherwise, if some replicators are offline, other replicators are contacted; if there are not enough online replicators, the transaction is aborted and should be repeated later, or it should require less on-line replicators. The number of replicators required to contact is a trade-off between the ease of completing the transaction and the probability that the update will eventually reach all replicators: suppose the owner required just 2 online replicators for the transaction; immediately after the transaction completes, the two replicators fail; the remaining replicators have the old version of the object and are not aware of the update.

The remaining replicators are notified about the update through the storage-layer messaging (see [35] for the complete description). The replicators that are currently on-line are contacted directly by the updated replicators: the updated replicator sends a simple notification consisting of the object ID and the address of the updated replicator. The replicators that are offline are contacted through persistent messaging: a message is send to the replicators' *synchro-peers*; once the replicator goes back online, it synchronizes with its synchro-peers, and gets the notifications. A replicator that is notified that some of its object are stale stops serving the stale objects until it gets the updated version from one of the updated replicators. By comparing the version IDs, the replicator is able to check whether a conflicting update happend.

4. STORAGE-LAYER LISTS

An online social network relies on a dialogue between its users; however, collaborative editing of an object, a storage layer proxy for a dialogue, is hard to support in a distributed file system using frequently-failing resources. In this section we describe *storage-layer lists (SLLs)*: a partially ordered list that Nebulostore implements at the storage layer (as a fundamental object type). A list is composed of *entries*: many users may *append* entries to a list owned and fully controlled by a single user (see section 5 for a more complete discussion on security aspects of our design). The notion of an entry with well-defined boundaries facilitates providing consistency in the presence of concurrent appends and temporal failures of replicators.

4.1 The storage-layer list (SLL) data structure

The storage-layer list is a collection of *entries* with a partial ordering. Each entry is a tuple containing:

- a key (a random integer);
- the predecessor entry's key;
- and the content (of a specified, short maximum length).

The predecessor's key is a concept similar to a logical clock; it enables to derive a tree-like ordering of the entries. This partial ordering can be used by the DOSN layer to infer temporal and/or causal relationships between entries in order to, e.g., render the conversation tree. When appending an element, the predecessor's key will be set to the "most recent" element as seen by the agent starting the append operation or the replicator that accepts the append (the meaning is chosen by the owner of the list, see the next subsection).

The list is stored using the list owner's replication space. In order to limit the possibility of free-riding, Nebulostore limits the size of the content of an entry. The default value should be enough for a text message or a small thumbnail image; in case of longer messages or binary entries (pictures, videos) the content of the entry contains the object ID `(bucketID, objectName)`; the object itself is stored on the replication space of the author of the entry. The owner of the list may change the maximum size of the entry; existing entries would be truncated or deleted (using the protocol for arbitrary modifications, section 3.4).

Even limited-size entries can be abused by malicious agents trying to deplete list owner's replication space by adding many entries. Such attacks are similar to the denial-of-service attacks (DOS) and can be thus similarly avoided by the replicators (by, e.g., increasing the delay between the subsequent requests from the same agent are served).

4.2 The protocol for the append operation

When an agent (not necessarily the owner) wants to add an entry to a list, it contacts one of the replicators of the list (section 3.2). The agent sends the content. Additionally, if the ordering is maintained by the agents, the agent also sends the key of the logically-previous entry as the predecessor key (as the agents can't be trusted, there is no guarantee that this rule will be followed by all agents: an agent may, e.g., decide to ignore all but the head element). Alternatively, the ordering can be maintained by the replicators: in this case, the replicator sets the predecessor's key to the most recent entry on the local copy of the list.

Next, the replicator picks a random key for the new entry and adds the entry to the local copy of the list.

Then, the replicator sends a message to all other replicators that a new element was added, along with the complete element. Similarly to the protocol used for propagation of arbitrary updates (section 3.4), direct (network-layer) messages are used for the on-line replicators and asynchronous messages for the offline replicators. To make appends cheap, we decided not to make the append a transaction-like operation (which would be similar to the whole protocol for arbitrary updates, section 3.4). Thus, it is possible that the replicator fails between accepting an entry from the agent and propagating it to other replicators (and the entry will be propagated only after the replicator returns online).

Finally, when a replicator gets a notification that an entry was appended, it adds the entry to the local copy of the list.

A similar protocol can be used for deleting a list entry (i.e., replacing an entry with a tombstone). The only additional requirement is that the contacted replicator must authenticate the agent as the author of the original entry (using, e.g., the agent's public key stored along the the entry).

5. SECURITY ON THE STORAGE LAYER

Within Nebulostore, each bucket has an owner who owns all objects stored in the bucket. The owner is also the party responsible for negotiating storage for the content (section 3.1). Access control within Nebulostore is kept as simplistic as possible in order to simplify the implementation, and in order to reduce the trust laid on replicators. The design intention is that protocols building on Nebulostore in turn implement more advanced access control via cryptographic means. We showcase one such design in our case study in section 6.

The access control in Nebulostore falls within the Discretionary Access Control (DAC) paradigm: the owner of a bucket decides the access control rules for all objects within the bucket. The following description concerns the storage-layer lists; standard (flat) objects support a subset of these operations. The access control is mainly specified via two boolean flags: *world/owner readable* and *world/owner appendable*. By default a newly created list will be owner readable and owner appendable, meaning that no-one but the owner can access the list at all. By setting these flags, the owner can allow any user who knows the list ID to read, and/or append entries to the list. Note that the right to perform arbitrary write operations is always limited to the owner alone.

All four combinations of these boolean controls are used. For lists which are private to the owner, they are both set to owner-only. On the other end, lists representing public files such as a wall in an online social network may be set to world-readable and world-appendable. Similarly, a list where only the owner publishes information would be world-readable but ownerwritable. Lastly, a list acting as an inbox or dropbox (allowing users to privately send messages to the owner) can be set up by making it owner-readable and world-writable.

To authenticate the owner, a cryptographic signature scheme is used. When the owner creates a bucket of objects (section 3.2) and negotiates replication agreements for it, he also creates a signature key pair and gives the signature verification key to replicators of the bucket. All operations which require authentication are signed using the owner's corresponding signature key, which he keeps private. The operations which are signed contain information such as to prevent replay attacks. This signature is verified by all replicators accepting the operation, including the replicators to which the update is propagated; in this way other replicators will reject (and perhaps report to the owner) non-authorized update forced by a rouge replicator.

In addition to these signatures on the protocol-level data, Nebulostore can also employ transport-layer encryption using TLS for all communication within the system. To defend against man-in-the-middle attacks, hosts in Nebulostore generate self-signed certificates. These certificates are stored in a DHT mapping from a Nebulostore identity to the certificate.

To prevent malicious modifications of the DHT by arbitrary nodes, the nodes storing values in this DHT are modified to only accept future write operations to a key if they are signed by the key corresponding to the certificate they store.

6. LISTS FOR DOSN USE CASES

In this section, we sketch a design for a DOSN building on Nebulostore. We detail only a few selected core building blocks of a whole DOSN system, these pieces being protocol-layer messaging, and wall functionality.

6.1 DOSN protocol-layer messaging

By DOSN protocol-layer messaging, we mean the functionality of being able to send messages on a protocol layer from one user (Alice) to another user she knows (Bob). These should not be confused with application-layer messages nor with storage-layer messages (used to coordinate object versions, section 3.4). The end user will typically not be aware of protocol-layer messaging occurring, except for some types which may result in dialogues or messages shown to a user. Examples of messages are: notification of a new wall post being posted on a friend's wall, a friendship request, or sending a cryptographic key.

Basic protocol-layer 1-to-1 messaging functionality can be built by having each user own an *inbox* list (not to be confused with an inbox for mail). We assume the address of this inbox is known to whoever wants to send messages to the user, for instance, it can be published as

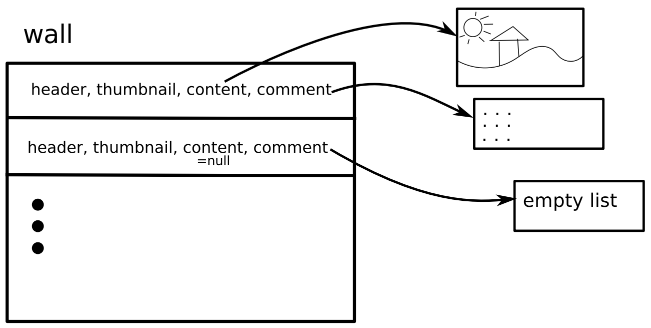
wall
header, thumbnail, content, comment
header, thumbnail, content, comment
=null
empty list

Figure 1: A wall using a Nebulostore list for the main content. The first entry contains the Nebulostore ObjectID of the complete content and an ObjectID of the Nebulostore list that will store the comments. The second entry is short (e.g., a text message), thus it does not refer to any external object.

part of the user's profile. The permissions on the inbox would be set to owner-readable and world-appendable. This allows any user who has the inbox ID (`(bucketID, objectName)`, section 3.2) to write messages to the inbox, but prevents users of the system from seeing how much traffic the inbox receives.

The owner will subscribe to the inbox while he is online, meaning that he will get notifications as soon as a message is received. Thus, this design provides for (close to) real-time performance while the user is online and persistance of messages while he is not, without the sender needing to adapt. By the list API, these message delivery and clearing functions are atomic. In addition, the list API also serves to provide write delineation between different messages. This is of importance in the presence of malicious users who may otherwise write e.g. partial messages to throw off parsing of later delivered messages.

To protect protocol-layer messaging, an address of the inbox is always published together with a public encryption key. The owner of the inbox holds the corresponding decryption key. When Alice wants to send a message to Bob, she encrypts the message using the key that Bob published. Alice would also sign the message using a signature key which is used to identify her in the DOSN.

6.2 Walls

Here, we show case how wall functionality can be implemented using lists. By wall, we mean a flow of events related to a particular user. This primarily includes events the user posts herself, in the form of status updates, images, or videos. Optionally, the user may also allow others to post information on her wall.

There are many different options in the literature for cryptographic access control of walls in a DOSN setting (see [8] for an overview). In this description, we simply assume that one of these cryptographic methods is used. Typically, with this type of access control, there is some header which allows authorized users to recover a symmetric key k, and from which unauthorized users cannot learn anything. The actual message posted on the wall can then be encrypted under k. We further make the assumption that the scheme also provides some integrity (e.g., by using a signature of posted content) such that the storage system or unauthorized users cannot tamper with messages or insert messages of their own.

Given these provisions, we can create a wall structure where walls are realized by each user creating a Nebulostore list for their wall. The access rights for this file would be world-redable, and world-appendable. Thus, even unauthorized users can append content to the wall, but the cryptographic integrity checks would fail on such content, and thus readers would ignore it and the owner will eventually garbage collect it.

Each post on the wall is posted into its own list entry (fig. 1). This entry contains, aside from the cryptographic parts previously discussed, a "thumbnail" of the message, an object ID (`(bucketID, objectName)`, section 3.2) of the full content (which may be null), and a Nebulostore-address of the comment thread. The thumbnail is the amount of information that the client needs to render the entry in a wall context. For images and videos, this is simply a thumbnail of the content. For text posts, this is usually the whole text of the message, but if the text is too long, the thumbnail only contains the first part. The address to the full content is the ID of an object in Nebulostore which contains the full content encrypted and integrity protected under the symmetric key k.

The comment thread is a "mini-wall" to contain all the comments on a list entry. Its format is very similar to that of the wall, with the addition of a special entry to indicate the parent post of a comment (supporting a thread structure of replies).

6.2.1 Friend Feed

Tightly related to the wall is the friend feed, which is a selection of the latest events posted by a user's friends on their walls. It is known from user studies [4] that the friend feed is one of the most visited views in social networks of today. How to efficiently implement a friend feed has been the subject of significant study both in the centralized and decentralized setting [21].

Approaches can be divided between push-based and pull-based approaches, as well as hybrids. In a push-based approach, the poster also pushes new posts to her friends so that the posts appear in their friends' friend feeds. In pull-based approaches, the user who wants to render a friend feed fetches up-to-date information from friends' walls.

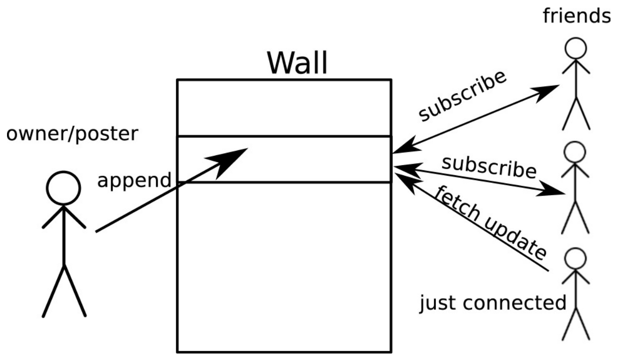

Figure 2: The friend feed. The two friends at the top-left corner are on-line and use the Nebulostore subscriptions to get the updates. The third friend has just connected to the network and fetches the updates using the standard read/download operation.

Building on Nebulostore, we propose a hybrid model building on the subscription feature. A user in our system will subscribe to the walls of all of her friends whenever she is online. When setting up the subscription, the user also fetches the latest list items from each wall, thus pulling content together to render the initial view. However, as long as her client is running, as she has subscriptions on all wall objects of her friends, any incoming changes will be pushed to her (using the Nebulostore subscription service, section 3.3). A benefit of this design (similarly to the 1-1 messaging) is that the poster is oblivious to whether writing her messages will result in data being pushed or pulled, keeping the protocol simple. We remark that it may be expensive to subscribe to all friends' walls if one has many friends, and thus one may want to intelligently pick a subset, or complement this system with other designs.

7. EXPERIMENTAL EVALUATION

We implemented a prototype of Nebulostore and tested it on PlanetLab (`www.planet-lab.org`). In this section, we briefly describe the implementation; then present the settings and the results of three performance evaluation tests: (i) to measure the storage space used by the two-level object IDs on the common resources, the space used by the DHT for the two-level addressing; (ii) to check the latency of the two-level object IDs, the time needed to access an object given its ID; (iii) to validate the list design, the time needed to propagate a list update to the replicas.

7.1 Nebulostore Implementation

Nebulostore prototype is implemented in Java 7 with a dependency on TomP2P (`tomp2p.net`) library that provides a Kademlia DHT. Internally, TomP2P uses

UDP to communicate between DHT nodes. We use standard TCP connections for object transfers and all Nebulostore internal messages.

The architecture of the system is modular and event-driven. Modules representing independent functionalities communicate via messages (sent locally or through the network). For thread management and message forwarding, we implemented a lightweight dispatcher.

Due to the modularity of the system, we were able to easily create a testing framework based on Nebulostore's internal messaging system. To each running instance of Nebulostore we are able to inject a testing module that simulates user file operations; the tests use the public API and follow a pre-defined scenario. After the test scenario is completed on every replica, results are sent to a selected test server and stored there for further analysis.

7.2 Storage space used by the two-level addressing

The goal of this test is to experimentally measure the size of the data stored in the DHT. We compare the two-level addressing (section 3.2) to a standard addressing, in which each object had an entry in the DHT (among others, PeerSoN [10] follows this approach).

As the type of the data store does not influence the obtained results, for this test we replaced the Kademlia DHT with a hash table stored at one of the nodes (implemented on a Berkeley DB); this allowed us to precisely measure the total used storage space.

During the test, 20 agents add 3000 objects to the system (each agent adds 150 objects); each object has 1KB. In case of two-level addressing, there are 20 buckets (one for each agent). In case of standard addressing, each object has an entry in the DHT. After each 200 objects, we measure the size of the resulting addressing layer.

Confirming the analysis in section 3.2, the storage space used by the two-level addressing is constant; counting other Nebulostore data, the device directory and the BerkeleyDB structures, the used space is around 212 kB.

7.3 Latency to access an object

Latency is one of the key factors affecting user experience. Current centralized OSNs use heavily optimized infrastructure, thus we assume that the increased latency will be the most significant drawback of DOSN.

To measure the average latency, we run Nebulostore on 100 PlanetLab nodes. We added up to 600 small (1KB) objects; we had 20 agents that picked an ObjectID from these 600 objects; follow the two-level address resolution protocol (section 3.2) to get the replicator's network address; and finally download the object from the replicator. We repeated 3 times the following test and computed the averages. In each phase of the test (a phase corresponding to the total of 20, 100, 200, 400, and 600 objects) each of the agents added objects; then broadcasted the ObjectIDs added in the phase (using the Nebulostore messaging layer); then picked a random ObjectID 5 times and performed 5 fetches for the address and 5 for the object.

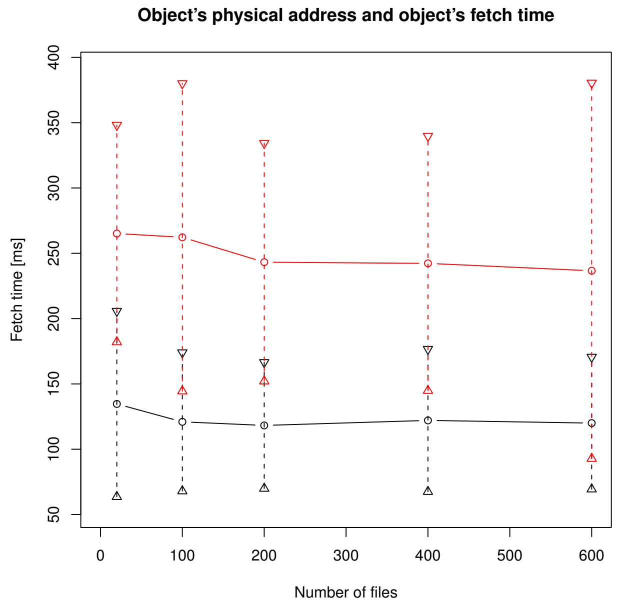

Figure 3: Experimental evaluation of the latency to get the object's physical address (black, bottom line) and then to download the object (red, top line). Network of 100 PlanetLab nodes with 20 active agents.

Due to problems with the DHT implementation, we were not able to use the Kademlia DHT for this test: we had simulated a DHT with a hash table stored at a single node. The DHT may result in additional delays. Moreover, Planet-lab nodes are not necessarily representative for consumer devices on which Nebulostore will be run: we expect the consumer devices to be less loaded, but also with lower-bandwidth Internet connection.

Our results are summarized in fig. 3. The average latency to get an object is roughly around 400 ms. Using commercial cloud computing providers, to download a 1KB file to PlanetLab nodes around the world, [18] reports average latencies between roughly 100ms (80% of requests for 3 fastest datacenters) and 500ms (all requests). Nebulostore is slower, but within the same range of values; we plan to optimize the fetch efficiency in the future.

7.4 Propagation time of list appends between replicas

The goal of this test is to measure the time a list entry

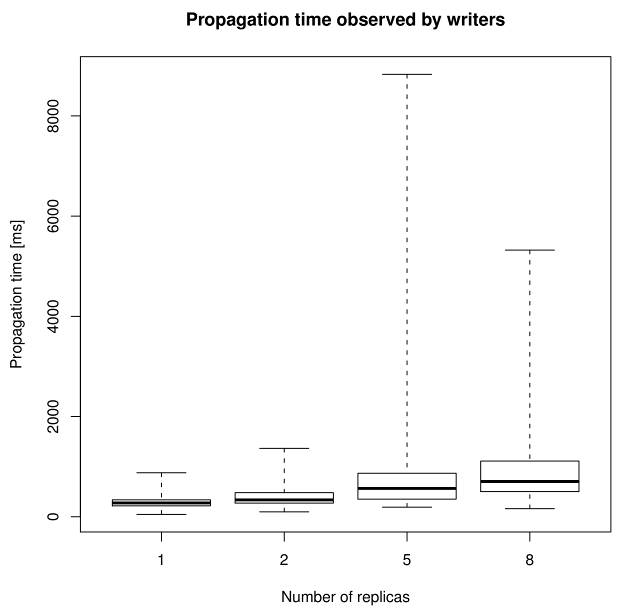

Figure 4: Experimental evaluation of propagation time; 100 PlanetLab nodes; average from 10 runs, 30 (1-2 replicas) and 5 (5-8 replicas) steps in each run. Measurements taken only by the writers. No outliers removed. The whiskers denote the minimal and the maximal values.

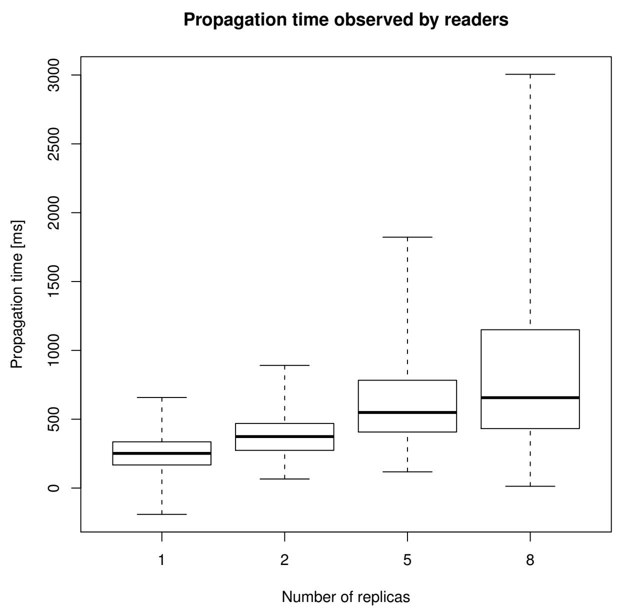

Figure 5: Experimental evaluation of propagation time; 100 PlanetLab nodes; average from 10 runs, 30 (1-2 replicas) and 5 (5-8 replicas) steps in each run; 10 readers for each replica. 5% of outliers removed. The whiskers denote the minimal and the maximal values.

added to one of the replicas takes to be propagated to other replicas.

We installed and ran the prototype on 100 nodes of the PlanetLab network. An agent creates a list (SLL) and sends its object ID to other agents: readers and writers-readers. The list is replicated on a specified number of replicators (in different runs: 1, 2, 5, 8). There are as many writers as there are replicas; each writer writes to a distinct replica. In total, there are 10 readers (taking into account both readers and writer-readers); each reader observes (reads from) one replica (the assignment is by round-robin). The following test run is repeated 10 times on different, randomly-chosen PlanetLab nodes. A single test run consists of 30 steps (1 and 2 replicas); or 5 steps (5 and 8 replicas). In each step, each writer appends a single entry to its assigned replica (writers act sequentially with sufficient time between subsequent writers to ensure that the write has propagated). Following the append protocol (section 4.2), the replicator propagates the entry to all other replicas. The appended entry consists of the address of the writer and its local physical clock. The readers continuously poll their replicas (a writer-reader reads from a different replica than the one he wrote to). When a reader detects a new entry on a replica, it saves the local physical clock and computes the propagation time as the time difference between the local physical clock and the physical clock value read from the entry.

This test relies on synchronization of physical clocks. We chose PlanetLab nodes that synchronize with an NTP server; however, for some nodes the differences between clocks are similar to the measured propagation times, rarely resulting even in negative propagation times. Rather than to ignore these measurements, we decided to take them into account so that they would balance exceedingly high measured propagation times (stemming from clocks skewed in the other direction). We thus assume that the average difference between physical clocks is zero and that the distribution is symmetric. To reduce the impact of extreme clock skews, we removed outliers (2.5% of highest and lowest measurements). To qualitatively assess the impact of these processing steps and the possible impact of clock skews, we present two figures: fig. 4 shows propagation times averaged without outlier removal and measured only by writers (so that there is no clock skew: a writer writes to one of the replicas and then reads from another replica); fig. 5 shows propagation times measured by both readers and writers and after removing outliers. We see that, although extreme values change, the medians and the quartiles are similar.

The propagation time is summarized in fig. 5. Although the propagation time increases with the increased number of replicas, the median for 8 replicas is less than 1 second. While this is not sufficient for real-time conversation, we believe that this is sufficient for asynchronous exchange of information using walls or inboxes.

8. CONCLUSIONS

We presented Nebulostore, a common storage layer for DOSNs. The main features of Nebulostore are: (i) the incentive-compatible design based on pair-wise storage contracts; and (ii) the list API that allows users to add content to other users' objects. We described how the lists can be used to implement common DOSN use cases, such as messaging, the wall and the friend feed. We implemented a prototype and tested it on PlanetLab; our tests show that common resources are used efficiently and that the propagation time is small.

Our results show that it is possible to achieve good availability (through incentive-compatible design rather than relying on altruism) and consistency (through a simple list-based API, rather than sophisticated filesystem operations). We have not demonstrated persistence — but only a large scale deployment and analysis of long-term user behavior can show whether the system is able to support itself during longer time periods.

A recent paper [24] doubted whether a large-scale distributed data storage or DOSNs are feasible. We believe that the advantages of organizational decentralization are too important to be defeated by technical difficulties. Distributed programming is intrinsically hard; however, lack of standard software components makes developing any complex distributed application — such as a DOSN — even harder. When developing a standard, centralized OSN, one does not expect to start with developing the operating system — instead, complex middleware is readily available. In contrast, almost the whole software stack is missing in case of decentralized applications.

We provide one such fundamental middleware component: a storage layer simple enough to be technically feasible, yet complex enough to support non-trivial applications. We hope that by using Nebulostore — both the design and the open-source implementation — other DOSN researchers, rather than starting from the scratch, will be able to focus their research on DOSNs.

9. REFERENCES

[1] Bernhard Amann, Benedikt Elser, Yaser Houri, and Thomas Fuhrmann. Igorfs: A distributed p2p file system. In *P2P, Proc.*, pages 77–78, 2008.

[2] M. Babaioff, J. Chuang, and M. Feldman. *Algorithmic Game Theory*, chapter Incentives in Peer-to-Peer Systems, pages 593–611. Cambridge, 2007.

[3] Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. Persona: an online social network with user-defined privacy. *SIGCOMM Comput. Commun. Rev.*, 39:135–146, August 2009.

[4] Fabrício Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virgílio Almeida. Characterizing user behavior in online social networks. In *ACM SIGCOMM Conference on Internet Measurements*, pages 49–62. ACM, 2009.

[5] R. Bhagwan, S. Savage, and G.M. Voelker. Understanding availability. In *IPTPS, Proc.*, volume 2735 of *LNCS*, pages 256–267. Springer, 2003.

[6] `http://labs.bittorrent.com/experiments/sync.html`, 2013. BitTorrentSync.

[7] Stevens Le Blond, Fabrice Le Fessant, and Erwan Le Merrer. Choosing partners based on availability in p2p networks. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(2):25, 2012.

[8] Oleksandr Bodriagov and Sonja Buchegger. Encryption for peer-to-peer social networks. In *Security and Privacy in Social Networks*, pages 47–65. Springer, 2013.

[9] William J. Bolosky, John R. Douceur, and Jon Howell. The Farsite project: a retrospective. *SIGOPS Oper. Syst. Rev.*, 41:17–26, April 2007.

[10] Sonja Buchegger, Doris Schiöberg, Le-Hung Vu, and Anwitaman Datta. PeerSoN: P2P social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, SNS '09, pages 46–52, 2009.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[12] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *International workshop on Designing privacy enhancing technologies: design issues in anonymity and unobservability*, pages 46–66, 2001.

[13] Leucio Antonio Cutillo, Refik Molva, and Thorsten Strufe. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine*, 47(12):94–101, 2009.

[14] Mojtaba Hosseini, Dewan Tanvir Ahmed, Shervin Shirmohammadi, and Nicolas D Georganas. A survey of application-layer multicast protocols. *IEEE Communications Surveys & Tutorials*, 9(3):58–74, 2007.

[15] Daniel Hughes, Geoff Coulson, and James Walkerdine. Free riding on gnutella revisited: the bell tolls? *Distributed Systems Online, IEEE*, 6(6), 2005.

[16] Sonia Jahid, Shirin Nilizadeh, Prateek Mittal, Nikita Borisov, and Apu Kapadia. DECENT: A

decentralized architecture for enforcing privacy in online social networks. In *PerCom Workshops*, pages 326–332, 2012.

[17] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35:190–201, November 2000.

[18] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: comparing public cloud providers. In *ACM SIGCOMM Conference on internet Measurement*, pages 1–14. ACM, 2010.

[19] Zhenhua Li, Jie Wu, Junfeng Xie, Tieying Zhang, Guihai Chen, and Yafei Dai. Stability-optimal grouping strategy of peer-to-peer systems. *Parallel and Distributed Systems, IEEE Transactions on*, 22(12):2079–2087, 2011.

[20] Matthew M Lucas and Nikita Borisov. Flybynight: mitigating the privacy risks of social networking. In *Proceedings of the 7th ACM workshop on Privacy in the electronic society*, pages 1–8. ACM, 2008.

[21] Giuliano Mega, Alberto Montresor, and Gian Pietro Picco. Efficient dissemination in decentralized social networks. In Tohru Asami and Teruo Higashino, editors, *Peer-to-Peer Computing*, pages 338–347. IEEE, 2011.

[22] Jean michel Busca, Fabio Picconi, and Pierre Sens. Pastis: A highly-scalable multi-user peer-to-peer file system. In *Euro-Par, Proc.*, 2005.

[23] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44, 2002.

[24] Arvind Narayanan, Vincent Toubiana, Solon Barocas, Helen Nissenbaum, and Dan Boneh. A critical look at decentralized personal data architectures. *arXiv preprint arXiv:1202.4503*, 2012.

[25] Shirin Nilizadeh, Sonia Jahid, Prateek Mittal, Nikita Borisov, and Apu Kapadia. Cachet: a decentralized architecture for privacy preserving social networking with caching. In *CoNEXT*, CoNEXT '12, pages 337–348, New York, NY, USA, 2012. ACM.

[26] Frédérique E. Oggier and Anwitaman Datta. Self-repairing homomorphic codes for distributed storage systems. In *INFOCOM*, pages 1215–1223, 2011.

[27] L. Pamies-Juarez, P. Garcia-Lopez, and M. Sanchez-Artigas. Enforcing fairness in p2p storage systems using asymmetric reciprocal exchanges. In *P2P, Proc.*, pages 122–131. IEEE, 2011.

[28] Lluis Pamies-Juarez, Frédérique E. Oggier, and Anwitaman Datta. Decentralized erasure coding for efficient data archival in distributed storage systems. In *ICDCN*, pages 42–56, 2013.

[29] R. Rahman, T. Vinkó, D. Hales, J. Pouwelse, and H. Sips. Design space analysis for modeling incentives in distributed systems. In *ACM SIGCOMM, Proceedings*, 2011.

[30] Krzysztof Rzadca, Anwitaman Datta, and Sonja Buchegger. Replica placement in P2P storage: Complexity and game theoretic analyses. In *ICDCS 2010, The 30th International Conference on Distributed Computing Systems, Proceedings*, pages 599–609. IEEE Computer Society, 2010.

[31] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):15–30, December 2002.

[32] Amre Shakimov, Harold Lim, Ramón Cáceres, Landon P Cox, Kevin Li, Dongtao Liu, and Alexander Varshavsky. Vis-a-vis: Privacy-preserving online social networking via virtual individual servers. In *COMSNETS*, pages 1–10. IEEE, 2011.

[33] Amre Shakimov, Alexander Varshavsky, Landon P Cox, and Ramón Cáceres. Privacy, cost, and availability tradeoffs in decentralized osns. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 13–18. ACM, 2009.

[34] Rajesh Sharma and Anwitaman Datta. Supernova: Super-peers based architecture for decentralized online social networks. In K. K. Ramakrishnan, Rajeev Shorey, and Donald F. Towsley, editors, *COMSNETS*, pages 1–10. IEEE, 2012.

[35] Piotr Skowron and Krzysztof Rzadca. Exploring heterogeneity of unreliable machines for p2p backup. In *HPCS*, 2013.

[36] Ral Gracia Tinedo, Marc Snchez Artigas, and Pedro Garca Lpez. Analysis of data availability in f2f storage systems: When correlations matter. In *P2P*, pages 225–236, 2012.

[37] Dinh Nguyen Tran, Frank Chiang, and Jinyang Li. Friendstore: Cooperative online backup using trusted nodes. In *SocialNet*, 2008.

[38] Kathleen D. Vohs, Nicole L. Mead, and Miranda R. Goode. The Psychological Consequences of Money. *Science*, 314(5802):1154–1156, November 2006.

[39] Zheng Zhang, Qiao Lian, Shiding Lin, Wei Chen, Yu Chen, and Chao Jin. Bitvault: a highly reliable distributed data retention platform. *SIGOPS Oper. Syst. Rev.*, 41:27–36, April 2007.