Department of
Information Engineering
and Computer Science
DISI

UNIVERSITY
OF TRENTO - Italy

DISI - Via Sommarive, 5 - 38123 POVO, Trento - Italy
http://disi.unitn.it

CLOUD-ASSISTED DISSEMINATION IN SOCIAL OVERLAYS

Giuliano Mega
Alberto Montresor
Gian Pietro Picco

August 2013

Technical Report # DISI-13-031

Cloud-assisted Dissemination in Social Overlays

***Abstract*—Decentralized social networks are an emerging solution to the privacy issues plaguing mainstream centralized architectures. *Social overlays*—overlay networks mirroring the social relationships among node owners—are particularly intriguing, as they limit communication within one's friend circle. Previous work investigated efficient protocols for peer-to-peer (P2P) dissemination in social overlays, but also showed that the churn induced by users, combined with the topology constraints posed by these overlays, may yield unacceptable latency.**

In this paper, we combine P2P dissemination on the social overlay with occasional access to the cloud. When updates from a friend are not received for a long time, the cloud serves as an external channel to verify their presence. The outcome is disseminated in a P2P fashion, quenching cloud access from other nodes and, if an update exists, speeding dissemination. We show that our protocol performs close to mainstream centralized architectures and incurs only modest monetary costs.

I. Introduction

Online social networks (OSNs) play a key role in the way we communicate over the Internet, as witnessed by the recent announce that Facebook has broken the psychological barrier of one billion active users. Unfortunately, this popularity is accompanied by concerns about privacy, amplified by some well-known incidents [8]. The root of the problem lies in the centralized architecture of mainstream OSNs, which requires the user to surrender control of sensitive personal data. As a consequence, interest in decentralized alternatives [14] has greatly increased in the last few years.

Decentralized OSNs and social overlays. In a decentralized OSN, users run clients on their machines. These clients form a peer-to-peer (P2P) overlay network collectively sharing and replicating content, and serving it on behalf of offline users when needed. Among the features offered by OSNs, arguably the most important is the ability to browse the profile of friends, and post updates to it. User profiles are the OSN equivalent of personal Web pages, to which both the owner and her friends can freely post *updates* as text and/or media [7].

Several proposals for P2P OSNs [9], [10], [18] rely on distributed hash tables (DHT) for content dissemination and storage. As a consequence, their operation ignores social relationships, and may result in security issues. In this paper, we are interested instead in approaches that rely on *social overlays*: overlay networks that mirror the social network. In these approaches [1], [12], [16], [21], communication is allowed only between two nodes whose owners are friends. This latter constraint is the key to improve, *by design*, privacy (non-friends do not see the information disseminated), locality (due to network homophily), and cooperation (due to friendship).

Limitations of social overlays. Similar to any other P2P system, those based on social overlays must cope with nodes joining and leaving of their own volition, according to user availability patterns known to be highly heterogeneous [22]. However, our previous work [2] has shown that, irrespective of the specific dissemination protocol, churn is a more severe concern in social overlays. A topology that could be easily repaired (e.g., using a DHT) may experience greater unreliability due to the smaller set of links available for reconfiguration (i.e., only those among friends). This results in large delays in the dissemination of profile updates that, according to [2], may exceed 3 hours for 1% of the receivers—unacceptable for a world-scale service like today's Facebook or Twitter.

P2P and the cloud: getting the best of both worlds. The problem of any purely decentralized dissemination protocol for social overlay is that the latter does not provide enough "options" to propagate updates in the presence of churn. Our idea is simple and yet effective: create an out-of-band channel that can "patch" connectivity if and when needed, by leveraging the persistence and ubiquity of cloud services.

In a nutshell, the scheme works as follows. The bulk of profile update dissemination is still carried out in a fully decentralized, P2P fashion on the social overlay. Specifically, we reuse the gossip-based protocol described in [1], due to its inherent simplicity. In addition, each node u is associated with a *profile store*, which is hosted on the cloud. Updates to u's profile (by u or some of its friends) are always performed first on the profile store, and then disseminated via the social overlay. Therefore, the profile store of u contains an always-available, consistent copy of its profile.

The availability of the profile store is key in overcoming the aforementioned delays in the propagation of profile updates. When a node v, friend of u, has not heard any update from u for a predefined time interval, it assumes that the update has been delayed, and verifies if this is the case by polling the profile store. In principle, this naïve solution is enough to overcome the limitations above. However, cloud access has a monetary cost, and we show that this solution has a poor performance/money tradeoff. We improve over this baseline by disseminating the outcome of polling the profile store back on the social overlay. This has the beneficial effect of quenching cloud access from other nodes (i.e., saving money) and speeding update dissemination. Results show that, compared to fully decentralized solutions, our hybrid one reduces maximum delays from hours to minutes, average delays from minutes to seconds, and places only a small cost to users.

Our approach inevitably reveals some information to the cloud provider—a limitation shared with other proposals [11]. Nevertheless, we argue that it is a significant departure from the *status quo* of centralized systems like Facebook. Indeed, the service agreement of the latter implies that user information is fully surrendered, while the service of cloud providers promises data confidentiality. Still, we share the belief of [11] that providers should be exposed to as little sensitive content as possible, and therefore keep the data in profile stores

encrypted, safeguarding our users from prying eyes.

Roadmap. The rest of the paper is organized as follows. Section II presents the system model, including our assumptions w.r.t. user availability, profile updates, and cloud access. Section III states the problem of mitigating dissemination delays over social overlays in the presence of churn. Section IV illustrates our hybrid solution leveraging the combination of P2P dissemination on the social overlay and occasional cloud access, which is then evaluated in Section V. Section VI places our work in the context of related efforts. Section VII draws conclusions and points at opportunities for future work.

II. System Model

We represent the social overlay as an undirected graph G, with $V(G)$ denoting its vertices, and $E(G)$ its edges. We define the *ego network* G_v of a node $v \in V(G)$ as the subgraph of G composed of v, the friends of v (its one-hop neighbors), and the edges among them. To simplify exposition we assume, without loss of generality, that there is a one-to-one mapping between nodes and users in the system. We therefore use the words "user" and "node" interchangeably.

Every user u in the network is associated to a unique profile page, which we denote as $\mathbf{pp}(u)$, that represents a typical OSN profile page, and might contain textual posts, pictures, comments from friends, among others.

We use a simplified model for the underlying physical network in which two nodes are able to communicate as long as they are online at the same time. We further do not model network transmission latencies since, as we will later see, these are relatively small w.r.t. to other sources of delay.

We assume clocks to be loosely synchronized, within the order of minutes. This can be easily achieved by NTP [20].

Node Availability. The P2P portion of our system, the social overlay, is composed of $|V(G)|$ nodes. Each node, at any point in time, is either logged in (online) or out (offline).

To drive this online/offline behavior, we adopt the well-known availability model of Yao et al. [22], henceforth referred to as the *Yao model*, in which an *alternating renewal process* is associated to each network node u. In the particular instance of the model we adopt, both the session lengths (online time between a login and the next logout) and the inter-session lengths (offline time between a logout and the next login) are exponentially distributed, with expected value 0.5 and 1.0 hours, respectively. In line with other works in the literature [19], we use exponential distributions instead of heavy-tailed ones because the latter would make simulations intractable. Heterogeneity is modelled as in [22].

Update Size. Although users share content of different nature, the vast majority of what is shared in the profile pages of modern OSNs are small objects under $150\ kB$, the average size of a Facebook picture. Such objects include text snippets, messages, and low-resolution pictures. While users also share larger objects, e.g. videos or high-resolution pictures, we argue that most of the times these are not part of profile pages themselves, but rather *linked* from third-party services such as YouTube or Flickr. Updates, therefore, are usually small, and concerns about latency take priority over bandwidth when gauging the quality of a dissemination technique.

Cloud Access. We assume the existence of a highly available, cloud-based service which nodes can access to store and retrieve data. Such a cloud service allows users to create personal storage areas under their control, i.e, they can selectively allow or deny read/write access to other users.

User profiles are relatively small (a few GB) and likely within the free quota currently allowed by some cloud providers, e.g., DropBox. However, for the sake of generality and to better elucidate the trade-off between delay and monetary cost, we adopt the *requester-pays* billing scheme of Amazon S3 [6], where users accessing data are charged for it. In other words, if a user v decides that she wants to download the new updates from her friend u directly from u's personal storage area (and is authorized to do so by u), it is up to v to pay for the download costs. This is important as it establishes the basis for the fair cost model of our solution: a user might opt to either go directly to the cloud and pay to immediately download updates from his friends, or use the free P2P network instead, possibly at a performance penalty.

In S3, costs can be broken down into three components:

- *Storage costs.* Keeping data in the cloud has a fixed monthly cost which increases with the amount of data stored, and is independent of whether or not it is accessed. These costs are very small in our case. First, profile pages consist of small-size content, so the total amount of data stored by a user should also be small (e.g., a few gigabytes). Second, storage is cheap: at the time of this writing, the yearly cost for 1 GB is around $1. Since storage costs do not impact our figures to a measurable extent, we choose to abstract them away altogether.
- *Bandwidth costs.* Similar considerations hold for bandwidth: updates are small and, as shown later, our protocol works by favoring many lightweight requests (a few hundred bytes per request) over fewer, larger requests.
- *Cost per request.* In S3, a user pays $0.01¢$ for every $10,000$ GET requests, or every $1,000$ PUT requests. We assume a similar cost model where read requests are cheaper than writes. The cost per read is the dominant one, and is therefore the one we focus on in this paper.

III. Problem Statement

The problem of *update dissemination* consists in diffusing, on a social overlay, copies of small updates posted by a *sender* node to a set of *receiver nodes*, with an acceptable delay.

A. Update Dissemination in Ego Networks

In an OSN, updates posted to a profile page $\mathbf{pp}(u)$ must be made available to all friends of u: when a friend of ours posts some content, we want to know about it. Less straightforward, perhaps, is to see that u is not the only one who can post new content to $\mathbf{pp}(u)$—indeed, any friend of u might originate new updates by, say, sharing new content in u's "timeline" (as in Facebook), or by posting comments to existing content. The

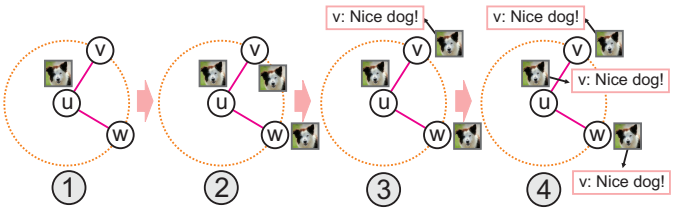


Fig. 1. User u posts a photo, user v posts a comment, user w just watches.

typical scenario we wish to support is depicted in Figure 1, where:

1) u starts by posting a picture to his profile page $\mathbf{pp}(u)$;
2) the system disseminates the update to the remainder of the ego network of u, namely v and w;
3) v sees the update and posts a comment to $\mathbf{pp}(u)$;
4) the comment once again gets disseminated to the remainder of the ego network of u, namely u and w.

Formally, let G_u be the ego network of some node u. We want to be able to disseminate updates from a sender node $v \in V(G_u)$ to all other receivers in $V(G_u)$. In particular, it could be that $v = u$, i.e., the sender is posting an update to its own profile, but $v \neq u$ is just as likely, e.g. when v replies to a previous post in the profile of u. Further, to reap the benefits of friend-to-friend cooperation, avoid spamming uninterested nodes, and avoid leaking updates to nodes who are not supposed to see them, we want to disseminate this update using only nodes from G_u itself.

B. Dissemination Delays and Churn

Disseminating updates with low latency over ego networks can be done efficiently when nodes are always online, as shown in [1]. When availability is taken into account, however, things get much more difficult. Friends may be rarely online at the same time, meaning that updates may be relayed across a chain of intermediate nodes before reaching their destination. This, in turn, may introduce *communication delays*.

The dynamics through which such delays arise is depicted in Figure 2 where, at time instant $t = 0$, node v starts disseminating an update over G_u. Since nodes a, u, and h are offline, the update cannot initially be disseminated beyond v himself. At time $t = 1$, node a comes online, allowing the update to flow over a path through a towards node d. At time $t = 2$ node h comes online, and the update flows to h, e and g. Finally, at time $t = 3$ node u comes online, and the update can reach the remaining nodes, completing dissemination.

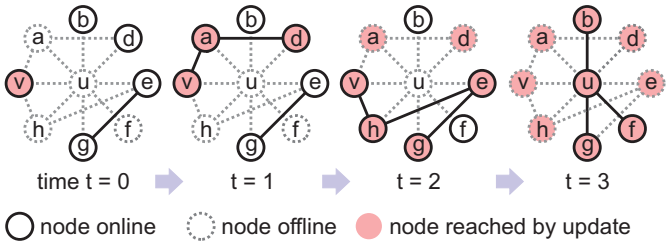
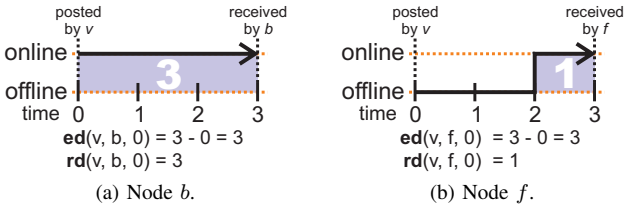


Fig. 2. Interplay between availability and dissemination delay.



(a) Node b. (b) Node f.

Fig. 3. Differences between $\mathbf{ed}$ and $\mathbf{rd}$.

C. Receiver Delay vs. End-to-End Delay

We consider two notions of delay. The first one is network-centric, defined as the total time elapsed from the instant t_0 at which an update is posted by a source v to the instant t_w at which it is received by a receiver w. We refer to this metric as the *end-to-end delay* from v to w w.r.t. t_0, or $\mathbf{ed}(v, w, t_0)$. It can be easily computed as $t_w - t_0$.

Being network-centric, $\mathbf{ed}$ is not representative of *user experience*. We capture the latter by measuring how long a user *had to wait online* before receiving an update. The intuition is that a receiver that logs in infrequently does not care if an update was posted a long ago (i.e., with a high $\mathbf{ed}$), provided it is received shortly after login. We refer to this metric as the *receiver delay* from v to w w.r.t. t_0, or $\mathbf{rd}(v, w, t_0)$.

These two notions of delay are compared in Figure 3, where we look more closely at nodes b and f of Figure 2: while $\mathbf{ed} = 3$ for both nodes, b had to wait online for longer. User experience was therefore better for f than it was for b, and this is reflected in the lower $\mathbf{rd}$ value.

D. Putting a Bound on Delays

As we have demonstrated in our previous work [2], relying on a purely P2P social overlay for disseminating updates over ego networks is not feasible in general. Delays can become unacceptably large, with average receiver delays of hours for a significant fraction (1%) of the nodes, and remaining above tens of minutes for over 10% of the nodes.

Our goal is to solve the problem of Section III-A while providing an acceptable user experience. Formally, we can express this goal as a *target delay bound* δ on update delivery. In other words, we would like to ensure that:

$$\mathbf{rd}(v, w, t_0) \leq \delta \tag{1}$$

for every source v, receiver w, and time instant t_0.

Putting a cap on $\mathbf{rd}$, however, is not enough, as it allows for some undesirable situations to emerge. Suppose we set δ to 20 minutes and, for the sake of argument, assume a receiver behaving as in Figure 4, logging in for 5 minutes every day. From the point of view of Equation (1), the bound is honored as long as we deliver the update before the end of the 4^{th} day.

But this is too long: a 4-day old update is not as useful as a 1-day old one. Further, the receiver did log in repeatedly during these 4 days, i.e., there were multiple windows of opportunity to deliver the update. This shows the main weakness of using a pure $\mathbf{rd}$ bound: it allows end-to-end delays to become excessively, unnecessarily large. We need a stronger bound, which can be established based on $\mathbf{ed}$. Yet, bounding $\mathbf{ed}$ directly is generally not possible. Figure 4 illustrates why:

if the receiver were offline at the instant when the bound is crossed, it would be impossible to deliver the update on time.

We reach a compromise by allowing a *soft delay bound* on **ed**. The idea is that as soon as the target delay bound δ is crossed, the system must deliver the update *at the next login of the receiver*. We express this by adding some slack time to the bound in case w is offline. This slack time represents the residual offline time $\mathbf{R}_{off}$ of w until its next login. Formally:

$$\mathbf{ed}(v, w, t_0) \leq \begin{cases} \delta & \text{if } w \text{ online at } t_0 + \delta \\ \delta + \mathbf{R}_{off} & \text{otherwise} \end{cases} \quad (2)$$

The slack time is a random variable, whose probability distribution results directly from the availability model. The bound in Equation (2), therefore, is no longer an exact, one-size-fits-all bound, but a probabilistic one exhibiting different statistical behavior for each receiver. This makes sense, since a different availability leads to different guarantees. Finally, note that since the slack is composed entirely of offline time, it does not count as receiver delay. Therefore, by honouring Equation (2) we are also automatically honouring Equation (1).

The goal of this paper, then, is providing a hybrid cloud/P2P dissemination protocol which can honor the soft latency bounds of Equation (2) while being efficient and low-cost.

IV. Cloud to the rescue!

The need for cloud resources arises because the social overlay cannot provide acceptable delays to all source/destination pairs in the network. As Figure 5 shows, this happens because the absence of certain nodes can create transient partitions that disrupt communication paths and introduce delay.

We need to "patch" such partitions somehow. A simple way to do so would be associating each node u to an *alias* $\tilde{u}$ of itself in the cloud. $\tilde{u}$ would ideally be indistinguishable from u to interacting nodes, and would be activated on-demand to satisfy requests on behalf of u, should u be offline. Such cloud aliases would effectively remove the transient partitions in Figure 5 and their associated delays, thus solving our problem.

Unfortunately this solution has several drawbacks. First, cloud providers that allow instances to be run on-demand (e.g., EC2 [5]) charge high prices per hour; running an alias for extended periods of time rapidly becomes economically unattractive. Hiring permanent, always-on hosting is also expensive, as lower-cost providers (e.g. [15]) charge more than $100 for one-year contracts. Second, we want avoid exposing the full memory state of the running software to the cloud provider, as this might reveal sensitive information (e.g. private keys). Finally, and most importantly, this solution requires that

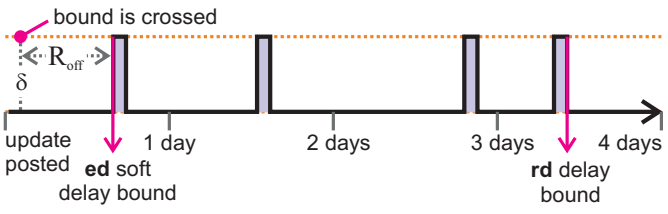


Fig. 4. Delay bounds for cloud-assisted dissemination.

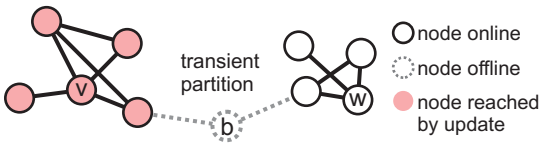


Fig. 5. Transient partitions in social overlay.

the nodes on the critical path of the transient partitions use and pay for an alias—not a feasible strategy in general.

A. Update Dissemination with Profile Stores

The aforementioned problems led us to an alternative solution where $\tilde{u}$ is no longer a full clone of u, but rather a simple high-availability *profile store* in which $\mathbf{pp}(u)$ is kept.

Publishing an update to a profile store is simple: if user v wants to post something to $\mathbf{pp}(u)$, it simply writes this update directly to $\tilde{u}$. Access control is also not a problem, since the primitives provided by services such as Amazon's S3 [6] make it straightforward to ensure that only authorized users get to write to $\tilde{u}$. By adopting the requester-pays model of S3, we ensure that each user has complete control over costs, as discussed in Section II. To minimize exposure of sensitive data to the cloud provider, all data stored in $\tilde{u}$ is encrypted before upload. This is in stark contrast with centralized solutions such as Facebook, whose business model effectively precludes storing encrypted data from being acceptable practice.

However, differently from cloud aliases, profile stores are passive in that they cannot initiate interactions with other nodes. Therefore, to actively overcome the transient partitions that cause delays, we resort to periodic *polling*.

Naïve approach. In its simplest form, our protocol works by having each node $w \in V(G_u)$ independently poll $\tilde{u}$ every δ time instants, retrieving any updates to $\mathbf{pp}(u)$ posted in the meantime. If w happens to be offline after δ time instants have passed, then w accesses the cloud immediately once it logs back in. This simple protocol, we refer to as PurePoll, allows us to circumvent the transient partitions of Figure 5 through an out-of-band channel, satisfying Equation (2).

Hybrid approach. While simple, PurePoll is wasteful in that it disregards the existence of low-delay paths in the social overlay which could be used to our advantage. To understand how, note that if we were to discard all paths in the social overlay that have delays above or close to the delay bound δ we wish to maintain—such as the paths that go through node b in Figure 5—we would be left with a set of disjoint groups for which the *internal* delays are low, as illustrated in Figure 6. We call these *delay groups*. To get a message disseminated over a set of delay groups while respecting the delay bound δ, all we have to do is to ensure that *at least one* node in each of these groups actually accesses the cloud every δ time instants. The other nodes can then get the update from this accessing node over the social overlay—which is fast enough inside the group—and avoid accessing the cloud themselves.

The second method we propose, named Hybrid, does just that. Each node $w \in V(G_u)$ keeps track of the last instant in time at which it heard any news from u. Whenever w goes for more than δ time instants without hearing from u, it polls

the profile store of u to see if there are new updates. If there are, w downloads them from the cloud and pushes them into the social overlay by means of an appropriate dissemination protocol— [1] in our case, whose specifics are not important. Otherwise, w pushes a special QUENCH message that contains the time t_0 at which w accessed the cloud and found nothing new. This message serves to inform other nodes that, as of t_0, there are no new updates, and that they can therefore refrain themselves from accessing the cloud for an extra δ time instants. We call this mechanism *access quenching*.

To see how HYBRID approximates the ideal situation of dissemination over delay groups we described previously, note that if two nodes belong to the same delay group then one will likely hear from the access of the other, resulting in access quenching. If two nodes do not belong to the same delay group, instead, it is unlikely that they hear each other in time to promote quenching, and two separate cloud accesses will ensue. Therefore, the protocol adjusts to the delay characteristics of the surrounding network, and provides a self-organizing mechanism for bridging transient partitions by polling.

Randomizing cloud accesses. A side effect of the protocol we described is that it induces nodes belonging to the same delay group to synchronize their accesses to the profile store.

The issue is illustrated in Figure 7a, in which node w_1 initially accesses $\tilde{u}$ at time t_0 and, having found no new updates, schedules its next access to $t_0 + \delta$ to respect the soft delay bound. At the same time, w_1 propagates this knowledge over the P2P network by means of a QUENCH message. Upon receiving the QUENCH message, node w_2 learns that, as of time t_0, there are no new updates to $\mathbf{pp}(u)$, and therefore it can refrain itself from accessing the cloud until $t_0 + \delta$. The two nodes are now synchronized. When we get to instant $t_0 + \delta$ (Figure 7a), the two nodes access $\tilde{u}$ at the same time. Having again found no updates, they disseminate their QUENCH messages, but to no effect: since the accesses are too close to one another, quenching is ineffective.

We address the problem by scattering access times near $t_0 + \delta$ as follows. First, we divide δ into a *fixed* component ψ and a *random* component α, such that $\delta = \psi + \alpha$. Then, let $\mathbf{T}$ be a random variable drawn from a uniform distribution $\mathcal{U}(0, \alpha)$. When node w_1 accesses the cloud and, later, when w_2 receives the QUENCH message, we now set their access times to $t_0 + \psi + \mathbf{T}$. This causes w_1 and w_2 to have slightly different access times after $t_0 + \psi$ which, provided α is sufficiently large, is enough to make quenching effective (Figure 7b). Since $0 \leq \mathbf{T} \leq \alpha$, this implies $\psi + \mathbf{T} \leq \delta$, i.e., the modified protocol still honours the soft bounds of Equation (2).

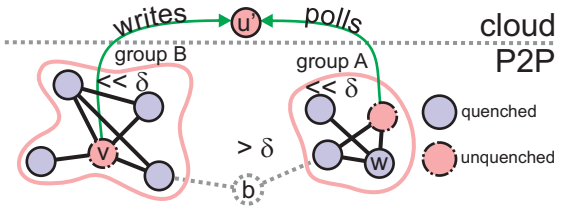
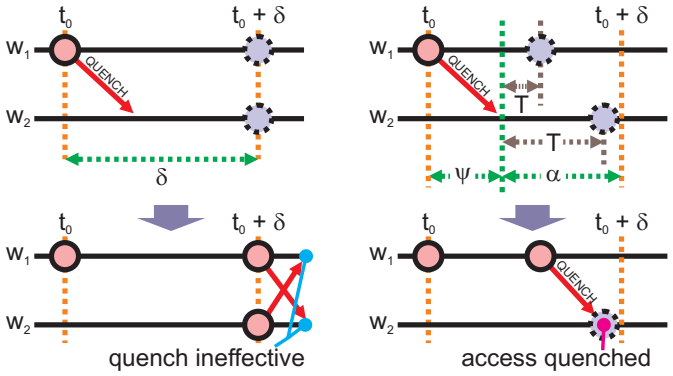


Fig. 6. Delay groups and HYBRID.



(a) Fixed. (b) Randomized.

Fig. 7. Quenching.

B. Hybrid in Detail

In HYBRID, every node $w \in V(G)$ keeps track of a *last seen* timestamp $last[u]$ which represents the last time instant at which w has heard any news from u. It also keeps track of a version number $version[u]$ for $\mathbf{pp}(u)$, used to determine whether its local version of $\mathbf{pp}(u)$ is up-to-date w.r.t. the one in the profile store $\tilde{u}$. In addition, w keeps track of its current, randomized target delay bound $target[u]$.

Whenever the target delay bound is crossed and w has not heard any news from u, it accesses the cloud. The description of the access protocol is given in Algorithm 1. The first action w takes, since it is about to acquire fresh first-hand information on the status of $\mathbf{pp}(u)$, is to suspend the dissemination of any QUENCH messages (lines 1–2), as these are effectively immediately stale. Next, w updates its last seen timestamp $last[u]$, as it is about to get the latest updates concerning u.

Then, w downloads all the identifiers of u's updates that are more recent than $version[u]$ (line 4). If U is not empty (line 5), $version[u]$ is set to the largest update version number contained in U, and all the updates that have not been received yet are downloaded from the cloud. The precise nature of the cloud operations depend on the API provided. For example, in S3 [6], this could be obtained through the versioning

Algorithm 1: OnTimeout

Trigger: Target delay bound is crossed (clock() − $last[u]$ > $target[u]$).

```
1  M ← P2P.query(⟨QUENCH, u, ·⟩)
2  forall m′ ∈ M do P2P.remove(m′)
3  last[u] ← clock()
4  U ← cloud.list(u, version[u])
5  if U ≠ ∅ then
6      version[u] ← max(U)
7      foreach id ∈ U − delivered do
8          upd ← cloud.download(u, id)
9          deliver(upd)
10         delivered ← delivered ∪ {id}
           P2P.disseminate(⟨UPDATE, u, last[u], id, upd⟩)
11 else
12     P2P.disseminate(⟨QUENCH, u, last[u]⟩)
13 target[u] ← ψ + uniform(0, α)
```

Algorithm 2: OnReceive

Trigger: Message is received from the P2P layer.
Input: Message m.

```
1  switch m.type do
2      case UPDATE
3          if m.id ∉ delivered then
4              deliver(m.upd)
5      case QUENCH
6          if m.t ≤ last[u] or clock() − m.t > δ then
7              P2P.remove(m)
8  if m.t > last[u] then
9      last[u] ← m.t
10     M ← P2P.query(⟨QUENCH, u, ·⟩)
11     forall m′ ∈ M : m′.t < m.t do P2P.remove(m′)
12     target[u] ← ψ + uniform(0, α)
```

mechanism and the "GET bucket object versions" primitive.

After download, w delivers the updates to the application layer (lines 9–10) and to the P2P dissemination layer (line 10), from where they are spread over the ego network G_u. As shown in line 10, update messages contain five fields: a type descriptor (UPDATE), the identifier of the owner of the profile page to which the update is addressed (u), the timestamp of when the update was downloaded (*last*$[u]$), the identifier of the update (*id*), and the update itself (*upd*).

If instead no updates are found (line 11), w disseminates a QUENCH message with the identifier of the profile page owner and the access timestamp (lines 11–12). Finally, w randomizes its target delay bound *target*$[u]$, as described in Section IV-A.

Upon receiving a message m, a node w executes Algorithm 2. If the message is an update (line 2) whose content is unknown to the receiver, it is delivered to the application layer (line 4). If m is a QUENCH message, and either its timestamp is older than *last*$[u]$ or the difference between the current local time and m's timestamp is larger than δ, the message is too old and its dissemination is interrupted (line 7).

Regardless of the type of message received, w updates *last*$[u]$ and its target delay bound whenever the timestamp of the message is more recent than its own—which will cause quenching at w—and stops disseminating any QUENCH messages with timestamp smaller than that of m (lines 10–11), since the knowledge contained in m is more recent.

Finally, it could happen that a node w joins the network being already timed out w.r.t. *target*$[u]$. In this case w would execute Algorithm 1. Yet, there could be some neighbor of w in the P2P network with recent updates on u, which would make cloud access unnecessary. To avoid this situation, in our simulations we ensure nodes *first* process incoming neighbor messages, if any, and then react to timeouts, if these still stand. In a practical implementation, a joining node could achieve the same effect by allowing some short grace period on login, thus giving neighbors enough time to send any new information.

V. EVALUATION

In this section we evaluate our protocol towards two goals: 1) provide supporting evidence that it performs significantly better than a pure P2P approach, and it is competitive with centralized approaches; 2) estimate what kind of monetary and network costs one should expect from running it.

A. Baselines

We compare our protocol against three baselines: *i)* PURE-POLL, which is the naïve protocol we described in Section IV; *ii)* PUREP2P, which disseminates updates exclusively over the social overlay, allowing us to measure the improvement brought by using the cloud; and *iii)* SERVER, which emulates a centralized approach akin to Facebook. This is achieved by adding a special server node which is connected to all the others, manages all profile pages, and can relay updates with zero delay. This represents the best performance reference for our system. Clearly, we wish to perform as close as possible to SERVER, while incurring only modest monetary costs.

B. Experimental Setting

Protocols are evaluated over a sample of 700 ego networks picked uniformly at random from the Orkut crawl of Mislove et al. [3]. The original graph contains 3 million vertices (i.e., 3 million ego networks), 223 million (undirected) edges, and has an average clustering coefficient of 0.171. Our sample includes around 5% of the vertices in the graph, and its average clustering coefficient is slightly smaller. A summary of statistics can be found in Table I. This is the same dataset we used in our previous work on social overlay delays [2]. For each ego network G_u in our sample, we pick *one* source node $v \in V(G_u)$ uniformly at random. This leads to the set of 700 (*source*, *ego network*) pairs we use for simulations. Let (v, G_u) be one pair. In a nutshell, we simulate updates originating at v, and measure delays towards receivers in $V(G_u)/v$, along with the metrics discussed in Section V-C. We then compute estimators (e.g., averages and empirical distribution functions), and base our discussions on those.

Our reliance on estimators motivates our choice to sparsely cover many ego networks (i.e., to pick 700 ego networks, and only one source within it) instead of covering fewer ego networks but more densely (e.g., to pick one big ego network, and 700 distinct sources in it). Since unbiased estimation relies on independent sampling, we want to insert as much variability as possible in the structure of the ego networks we study, to ensure that the values of metrics are uncorrelated.

We then repeat, for each (*source*, *ego network*) pair, the following experiment 100 times. First, we set all nodes in G_u to offline and, for PUREPOLL and HYBRID, we also set all the values of *last*$[u]$ (the last time instant at which a node heard from u) to zero. Since this initial state is not representative of

Metric	Value
Number of Vertices	137892
Number of Edges	1460043
Average Clustering Coefficient	0.112
Average Egonet Size	197.5
Maximum Egonet Size	2181

TABLE I
STATISTICS FOR THE EGO NETWORK SAMPLE USED IN OUR EXPERIMENTS.

the steady-state regime of our system, we run the simulation for a *burn-in period* γ in which no measurement is taken. During burn-in, we simulate the churn model and, in the case of PUREPOLL and HYBRID, we also simulate the polling of the cloud and the propagation of QUENCH messages.

The goal is to obtain a representative online/offline configuration for the network before we start injecting updates, but also a representative number of QUENCH messages going around and a representative "phase shift" for the last seen timestamps *last*$[u]$ at the various nodes. The nodes are initially all synchronized at *last*$[u] = 0$ but, as the burn-in progresses, they access the cloud at different rates due to their different inter-session lengths, causing their values of *last*$[u]$ to become different, particularly for nodes in different delay groups. We have empirically established, with heuristics similar to [4], that setting γ to 48 hours is enough to produce unbiased results.

After burn-in, the experiment progresses by waiting for the first login of the source v, at which point we cause v to post an update to $\mathbf{pp}(u)$. To avoid having to deal with the complexities of building a user model—which would not buy us much in any case for the kind of performance and cost measurements we do—we choose to be pessimistic and assume that nodes always post their updates at beginning of one of their sessions (the earliest time instant when a user can post).

The posting of the update by the source marks the beginning of our *measurement session*, which proceeds until the update reaches all destinations in $V(G_u)/\{v\}$. At that point, the experiment ends. To enable a more precise discussion of the metrics we compute, described next, we represent the set of $n = 100$ independent experiments we run for an ego network G_u with source node v as $S(G_u) = \{e_1, \cdots, e_n\}$. The measurement session of experiment e_i starts at time instant s_i and ends at time instant f_i, and has *duration* $\mathbf{d}(e_i) = f_i - s_i$.

C. Metrics

Delay. We focus on the *average* end-to-end delay, which we refer to as $\mathbf{aed}$, and the average receiver delay, $\mathbf{ard}$. Each experiment $e \in S(G_u)$ generates exactly one $\mathbf{ed}$ and one $\mathbf{rd}$ delay sample per source/destination pair (v, w) in G_u. If we denote the $\mathbf{ed}$ sample for pair (v, w) generated by experiment e as $\mathbf{ed}(v, w, e)$, we can compute $\mathbf{aed}$ as the sample average:

$$\mathbf{aed}(v, w) = \frac{1}{|S(G_u)|} \sum_{e \in S(G_u)} \mathbf{ed}(v, w, e)$$

and $\mathbf{ard}$ can be computed with a similar formula.

Monetary cost. We measure the yearly costs of running the system by counting, for each node $w \in V(G_u)$ and each experiment $e \in S(G_u)$, the total number of times $\mathbf{cs}(w, e)$ that w accesses the cloud over the measurement session of e. We define the *average access rate* of w over G_u as:

$$\mathbf{acs}(w, G_u) = \frac{\sum_{e \in S(G_u)} \mathbf{cs}(w, e)}{\sum_{e \in S(G_u)} \mathbf{d}(e)} \frac{\text{access}}{\text{hour}}$$

As mentioned in Section II, we assume that the dominant cost is the price of GET requests, identified by ρ. The yearly cost incurred on node w for keeping up-to-date with the updates of one profile page $\mathbf{pp}(u)$ can be therefore approximated by:

$$\mathbf{ycs}(w, u) \sim \rho \times \mathbf{acs}(w, G_u) \frac{\text{access}}{\text{hour}} \times \left(24 \frac{\text{hour}}{\text{day}} \times 365 \text{ day}\right)$$

To get an estimate of the overall yearly spendings $\widehat{\mathbf{cs}}(w)$ of w, we would need to estimate and sum the yearly costs incurred by w to keep up-to-date with each of the friends in $V(G_w)/\{w\}$, i.e., $\sum_{m \in V(G_w)} \mathbf{ycs}(w, m)$. In practice, this means computing estimates for 137 892 ego networks, which is intractable given the high costs of these simulations.

We therefore choose to use two distinct approximation models for costs. These are less accurate, but also much less expensive to compute. Both models can be expressed as the product of $\mathbf{ycs}(w, u)$ by an approximation constant $\tau(w)$:

$$\widehat{\mathbf{cs}}(w) \sim \mathbf{ycs}(w, u) \times \tau(w)$$

The models differ in how we compute $\tau(w)$. In the simplest model, which we call the *flat approximation*, $\tau(w)$ is simply the average size of the ego networks in our sample, i.e., we multiply $\mathbf{ycs}(w)$ by $\tau(w) = 197.5$, for all w. This applies a fixed "penalty" to all nodes, regardless of the conditions of the surrounding ego networks, or their number. In the second model, called *degree approximation*, $\tau(w)$ is instead the degree of w in the social network. This assumes that costs increase linearly w.r.t. node degree. In particular, it makes the assumption that all friends of w incur the same yearly costs as w, and that the total cost can be computed as their sum. Intuitively, these approximations yield similar estimates for nodes whose degree is close to the average, but differ significantly for nodes with very low and very high degrees.

Given our fixed computational budget, these cost models are a necessary tradeoff between obtaining unbiased estimates for delay (requiring that we cover many uncorrelated ego networks) and accurate estimates for cost (requiring that we densely cover neighboring, hence correlated, ego networks). Since our focus is on delays, we choose to be precise with the former, while settling for less accurate estimates for the latter.

Network cost. To assess the usage of network resources, we measure the number of messages a node processes per time unit, on average, to keep up with updates from its friends. Similarly to what we did for cloud accesses, let $\mathbf{msg}(w, e)$ represent the number of messages processed (sent/received) by w during the measurement session of experiment e. The average hourly rate $\mathbf{amsg}(w, G_u)$ at which w processed messages, therefore, is given by:

$$\mathbf{amsg}(w, G_u) = \frac{\sum_{e \in S(G_u)} \mathbf{msg}(e, w)}{\sum_{e \in S(G_u)} \mathbf{d}(e)} \frac{\text{message}}{\text{hour}} \quad (3)$$

Again, this gives us the costs incurred on w while keeping up-to-date with a single friend. We adopt a similar approximation as we did with $\widehat{\mathbf{cs}}$ when computing the total message processing rate, and multiply $\mathbf{amsg}$ by $\tau(w)$:

$$\widehat{\mathbf{msg}}(w) \sim \mathbf{amsg}(w, G_u) \times \tau(w) \quad (4)$$

We again use both the flat and the degree approximations when computing τ, observing the same caveats as before.

D. Results

We use the shorthand notation HYBRID/ψ/α to refer to the variant of HYBRID with parameters ψ and α, with unit given in minutes. We use a similar notation, PUREPOLL/δ, to refer to the PUREPOLL variant with target delay bound δ.

We simulate two versions of HYBRID, HYBRID/30/14, and HYBRID/15/14. These parameter settings, as we later show, provide good performance in terms of delay and cost, and we use them for comparison against the baselines. However, as explained in Section IV, the target delay bound of HYBRID is randomized, and varies in $[\psi, \psi + \alpha]$ with average $\psi + \alpha/2$. Since PUREPOLL is not randomized, we compare each setting of HYBRID to three settings of PUREPOLL: *i) "fast"*, PUREPOLL/ψ; *ii) "intermediate"*, PUREPOLL/$(\psi + \alpha/2)$; *iii) "slow"*, PUREPOLL/$(\psi + \alpha)$.

This yields six PUREPOLL variants: three ($\delta \in \{30, 37, 44\}$) to compare against HYBRID/30/14 and three ($\delta \in \{15, 22, 29\}$) to compare against HYBRID/15/14. Since PUREPOLL/30 and PUREPOLL/29 behave essentially the same, we do not show the former and use the latter. Finally, to understand at which point PUREPOLL can overtake HYBRID, we add a seventh setting for PUREPOLL in which we set $\delta = 5$.

Delay. Figure 8 shows cumulative distribution functions (CDFs) for receiver and end-to-end delays of HYBRID and baselines. Since **rd** is always zero for SERVER, we omit it from the plot. Complementary statistics are given in Table II.

The data confirms that PUREP2P suffers from significant performance issues, with the **ard** distribution having a long tail, reaching values as high as 2.9 hours, and remaining nevertheless above 1.4h at the 99^{th} percentile. We repeat the observation of [2] that this small 1% can translate into a bad experience for millions in a system of the scale of Facebook.

Further, the data shows that our cloud-based alternatives—HYBRID and PUREPOLL—effectively *solve the problem of the long delay tail* by putting a bound on **rd**, which can be seen from the much smaller maximum and 99^{th} percentile and maximum values in comparison.

Finally, we see that HYBRID outperforms its associated PUREPOLL variants (including the fast one), while providing a better experience for a significant fraction of the users across all parameter settings, even as we compare HYBRID/30/14 to PUREPOLL/5. By combining both approaches, HYBRID effectively reconciles the best of both worlds: the fast performance of PUREP2P for the regions of the network that exhibit low delay—which can be seen in Figure V-D as the nearly vertical shape of the CDF up until the 60^{th} percentile—with the ability of mitigating the long delay tails of PUREPOLL.

Table II shows that HYBRID has maximum and 99^{th} percentile **ard** values which are comparable to those of their *fast* PUREPOLL counterparts (i.e., HYBRID/ψ/α achieves performance similar to that of PUREPOLL/ψ), with HYBRID being nevertheless faster. Indeed, HYBRID/30/14 and HYBRID/15/14 perform around 837% and 685% percent faster, on average, than PUREPOLL/29 and PUREPOLL/15, respectively.

Figure 8b shows that HYBRID significantly improves end-to-end delays as well, particularly at lower percentiles. Again, these reductions are for nodes connected by low-delay paths in the overlay. The polling period, never smaller than δ, represents a barrier to PUREPOLL that does not exist for HYBRID. Finally, HYBRID is much closer to SERVER than PUREP2P.

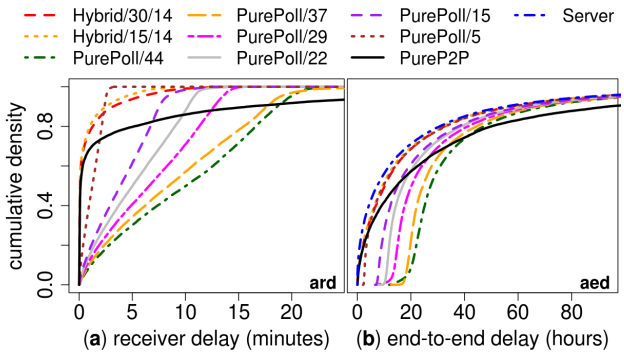


Fig. 8. CDFs for **aed** and **ard** for all source/destination pairs.

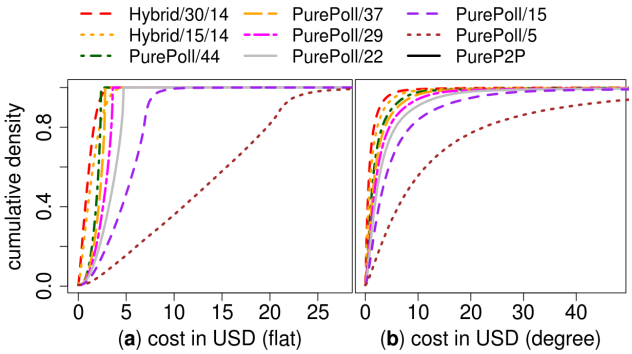


Fig. 9. Yearly monetary cost estimates for HYBRID.

The higher percentiles, instead, are dominated by the behavior of low availability nodes—i.e., nodes that stay offline for extended periods of time—causing **aed** distributions to converge, as evidenced by the diminishing distances of curves in Figure 8a at higher percentiles. Since values are similar for all protocols, we omit them from Table II. The maximum **aed** is, in any case, around 2 days, with the 99^{th} percentile at about 3 hours, for all approaches—including SERVER.

Monetary cost. Yearly cost figures are shown in Figure 9. We use the current Amazon S3 pricing [6], as per the model of Section II. Additional statistics are provided in Table III.

	ard		
	avg.	99^{th}	**max.**
PUREP2P	5.76m	1.5h	2.9h
HYBRID/30/14	48s	9.8m	15.2m
HYBRID/15/14	35s	7.2m	13.5m
PUREPOLL/44	9.94m	22.9m	27.1m
PUREPOLL/37	8.9m	21.6m	26.4m
PUREPOLL/29	6.7m	14.5m	16.6m
PUREPOLL/22	5.2m	11.9m	14.3m
PUREPOLL/15	4m	10.7m	12.3m
PUREPOLL/5	1.2m	2.8m	4.5m
SERVER	0	0	0

TABLE II
DISSEMINATION DELAY (m = MINUTE, s = SECOND, d = DAY).

Again, the costs for HYBRID are generally lower than their associated PUREPOLL variants. Costs are extremely attractive under the flat model, with HYBRID/15/14 presenting a very good cost/latency tradeoff. The other side of the coin is given by the degree model, which reaches high maximum values. To understand what is going on, however, we need to take a closer look. Figure 10 presents scatter plots of estimated yearly costs versus node degrees for both HYBRID variants. We can see that high values all originate from a small set of extremely connected nodes, the most connected having 33 313 friends. For nodes with less than 1 000 friends, however, costs for HYBRID/30/7 and HYBRID/15/7 are no larger than $15 and $21 a year, respectively, which are still significantly below the $127 price tag of low-cost hosting solutions [15], or the minimum of $105 required to keep an Amazon EC2 micro instance (the cheapest available) running for a year [5].

Even if some users do get a large number of friends, we still do not expect to see these high costs in a real-world setting. Users with huge ego networks are likely to trim friends they do not want to keep in touch so often, bringing down costs considerably. And that is precisely the strong point of our solution: the user can decide whether and how much to pay, as well as which latency bounds to keep with which friends.

Network cost. The last metrics we present regard the usage of P2P network resources. We analyse the average number of messages processed per second at each node, using the same flat and degree approximations as before to produce the overall estimates. We focus on QUENCH messages, since our main interest is on the base cost of the protocol. Indeed, the cost incurred by updates depends on user posting frequency and habits, variables out of our control. Yet, we argue that evaluating only QUENCH overhead is reasonable, since the bandwidth available for updates is ultimately given by what is available, minus the overhead of our protocol measured here.

Statistics for message costs are presented in Table IV. The two protocols present very similar performance. This reflects the fact that the QUENCH mechanism causes the underlying push protocol to work continuously in both cases. The network overhead, therefore, is similar for both variants, and should remain so for any variant with smaller values of ψ and α.

If assume the size of a QUENCH message to be around 48B (a 40B TCP/IP header plus 64 bits for identifier and timestamp), the overhead of running HYBRID is reasonable for a significant percentage of the nodes. Indeed, even under the pessimistic degree approximation, 90% of the nodes have to process less than 40 msg/s, which translates into running costs of around 2kB/s, without any optimizations (e.g., aggregation of ids and timestamps under a single message). For the flat model, these costs are even lower, with 99% of the nodes having to process less than 70 msg/s, or around 3kB/s.

Maximum values are still reasonable under the flat approximation, reaching at most 178 msg/s, or around 8.5kB/s. Under the degree approximation, on the other hand, these clearly become unreasonably large. The high numbers are again mostly due to nodes with high connectivity. This phenomenon can be observed both in Table IV, where we take the degree of the node at the n^{th} percentile for both protocols, average them, and display them on the top row; and in the scatter plots of Figure 11, where we see that very few nodes with degree less than 1 000 would have to process more than 100 msg/s (the equivalent of 4.8kB/s by our previous estimates). Indeed, under the flat approximation, only 0.02% of the nodes go above 100 msg/s, and 3% under the degree approximation.

We further note that the degree approximation is pessimistic under another important aspect, namely, it assumes a worst-case scenario in which a node w is concurrently participating in the dissemination of QUENCH messages for all of its friends, i.e., all the profile pages $\mathbf{pp}(u)$, $u \in V(G_w)$. Using the availability theorem of [22], however, we can predict that this

	flat (USD)			degree (USD)		
	avg.	99^{th}	**max.**	**avg.**	99^{th}	**max.**
HYBRID/30/14	1	2.9	3.4	1.25	9	230
HYBRID/15/14	1.42	3.9	4.66	1.72	12.4	304
PUREPOLL/44	1.84	2.38	2.9	2.5	17.4	282
PUREPOLL/37	2.13	2.8	3.47	2.9	20.3	324
PUREPOLL/29	2.63	3.59	3.81	3.57	25	390
PUREPOLL/22	3.32	4.72	4.88	4.51	31.6	483
PUREPOLL/15	5	9.5	20	9.5	48	737
PUREPOLL/5	13	27	64	17	128	1909

TABLE III
YEARLY COSTS, IN US DOLLARS.

	avg.	90^{th}	95^{th}	99^{th}	**max.**
node degree	197.5	409	658	2 897	33 313
HYBRID/30/14 (degree)	25.98	43.17	77.9	298.16	9 498
HYBRID/15/14 (degree)	26.02	43.17	77.6	294.73	9 472
HYBRID/30/14 (flat)	11.64	32.60	46	73.2	178
HYBRID/15/14 (flat)	11.8	33	56	74.8	168

TABLE IV
NETWORK COST, IN QUENCH MESSAGES PER SECOND.

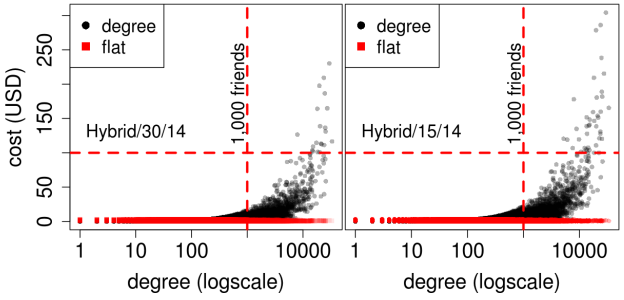


Fig. 10. Degree vs. monetary cost scatter plots for HYBRID.

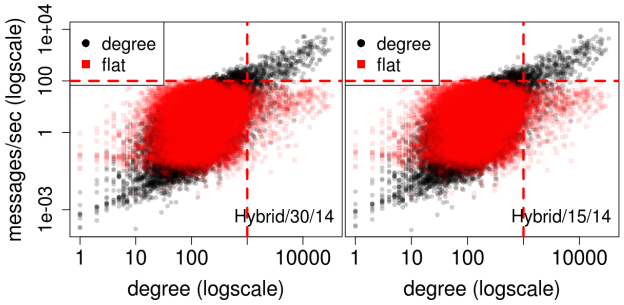


Fig. 11. Degree vs. network cost scatter plots for HYBRID.

is a rare situation in practice. Under the availability settings generated by the model of Section II for our experiments, the theorem predicts that, on average, only around 39% of the nodes would be online at any given point in time, meaning that these values should be much lower.

In any case, network costs have great margins of optimization, due to our choice of a pure push approach [1], known to tradeoff latency for overhead. An alternative would be to use this approach in a push-pull, anti-entropy protocol [13]. We believe this would significantly reduce network costs, while maintaining the good performance w.r.t. the other metrics.

VI. Related Work

P2P Decentralized OSNs. A number of proposals appeared in the past few years [9], [10], [18]. Although these are based on overlays different from the one we propose here, these systems also require the ability to asynchronously multicast profile updates to friends. The standard technique, in these cases, is to have nodes contact their friends and push messages either directly (which invariably leads to performance problems) or by storing them in a DHT. In the latter case, messages are stored at a random location known to the receiver. Our technique could offer added performance and reliability guarantees for such asynchronous message transfers, while improving privacy by the use of a known, authorized third-party to store updates, as opposed to randomly assigned DHT nodes.

Social overlays. There has been a growing interest in the use of social networks as communication networks, and social overlays have made their way into a number of system proposals over the past few years [1], [12], [16], [21]. Commonly cited reasons for using social overlays include desirable security properties, anonymity, censorship-resistance, among others. Given that these networks are inherently susceptible to partitioning under churn, the work we present here represents an important step in rendering such systems practical.

Cloud-assisted P2P and OSNs. The idea of using a cloud infrastructure to boost performance of P2P is not new. Cloud helpers are proposed in [17] to increase availability in a friend-to-friend (F2F) backup system. This is similar to our work in that F2F systems are also based on social overlays, and that data is also confined to ego networks. The problems are different, however, in that backup is concerned with providing high availability to the user backing up her data, and only to herself. Data gets updated, but time requirements are less stringent. Finally, backup data is generally larger than our objects, shifting concerns to throughput instead of latency.

Confidant [11] directly targets OSNs by relying on cloud aliases. Data is kept only at the P2P layer, replicated only among friends, while aliases act as coordinators tracking replicas and membership (i.e., who replicates what, and who is online). In this setting, churn becomes a challenge to the availability of data, instead of its dissemination. Our approach makes opposite choices w.r.t. the placement of data and control, therefore exploring a different set of design tradeoffs.

VII. Conclusions and Future Work

Social overlays are an interesting option for building decentralized OSNs, but their susceptibility to transient partitioning under churn renders key functionality such as fast dissemination of profile updates difficult to implement efficiently. In this paper, we have presented a solution to these inefficiencies, by introducing a protocol that leverages on a highly available cloud infrastructure to adaptively support the overlay when and where needed, without sacrificing the fundamental property of allowing communication only among friends.

Our results show that dissemination delays can be dramatically improved and monetary costs limited for users with less than one thousand friends. However, network costs, albeit acceptable, can be optimized further. We intend to address this issue in future work by investigating a combination of the current push approach with an anti-entropy mechanism.

References

[1] Reference removed due to double-blind review.
[2] Reference removed due to double-blind review.
[3] A. Mislove et al. Measurement and analysis of online social networks. In *Proc. Internet Measurement Conf.*, 2007.
[4] C. Alexopoulos and A. F. Seila. Implementing the batch means method in simulation experiments. In *Winter Simulation Conference*, 1996.
[5] Amazon EC2 website. http://aws.amazon.com/ec2/, [March 2013].
[6] Amazon S3 pricing. http://aws.amazon.com/s3/pricing/, [March 2013].
[7] D. Boyd and N. B. Ellison. Social network sites: Definition, history, and scholarship. *Jrnl. of Computer-Mediated Comm.*, 13, 2007.
[8] D. Boyd and E. Hargittai. Facebook privacy settings: Who cares? *First Monday*, 15(8), 2010.
[9] S. Buchegger, D. Schiöberg, L.-H. Vu, and A. Datta. PeerSoN: P2P social networking – early experiences and insights. *Proc. Workshop on Social Network Systems (SNS'09)*, 2009.
[10] L. A. Cutillo, R. Molva, and T. Strufe. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications*, 47(12):94–101, Dec. 2009.
[11] D. Liu et al. Confidant: Protecting OSN Data without Locking it Up. In *Proc. Conf. Middleware*, 2011.
[12] A. Datta and R. Sharma. GoDisco: Selective gossip based dissemination of information in social community based overlays. In *Proc. of Intl. Conf. Distributed Computing and Networking (ICDCN'11)*, 2011.
[13] A. Demers et al. Epidemic algorithms for replicated database maintenance. In *Proc. Intl. Conf. on Principles of Distributed Comp.*, 1987.
[14] Diaspora website. https://joindiaspora.com/, [March 2013].
[15] Dreamhost website. http://dreamhost.com/, [March 2013].
[16] Freenet website. http://www.freenet.org, [March 2013].
[17] R. Gracia-Tinedo, M. Sanchez-Arigas, and P. Garcia-Lopez. F2box: Cloudifying f2f storage systems with high availability correlation. In *Proc. IEEE Conf. on Cloud Computing*, 2012.
[18] K. Graffi, C. Gross, P. Mukherjee, A. Kovacevic, and R. Steinmetz. LifeSocial.KOM: a P2P-Based platform for secure online social networks. In *Proc. Conf. P2P Computing (P2P'10)*, 2010.
[19] A. Singh, G. Urdaneta, M. van Steen, and R. Vitenberg. Robust overlays for privacy-preserving data dissemination over a social graph. In *Proc. Intl. Conf. Dist. Computing Sys. (ICDCS)*, 2012.
[20] A. Tanenbaum and M. van Steen. *Distributed Systems—Principles and Paradigms*. Prentice Hall, 3rd edition, 2007.
[21] Tonika website. http://5ttt.org, [March 2013].
[22] Z. Yao et al. Modeling heterogeneous user churn and local resilience of unstructured P2P networks. In *Proc. Intl. Conf. Network Protocols (ICNP'06)*, 2006.