Social Overlays Meet the Cloud: A Hybrid Architecture for Profile Dissemination in Decentralized Social Networks

Giuliano Mega, Alberto Montresor, *Member, IEEE,* and Gian Pietro Picco, *Senior Member, IEEE*

Abstract—Decentralized online social networks (OSNs) have attracted the attention of researchers and practitioners as a viable alternative to the current centralized OSNs, whose business model is based on the exploitation of the private data of their users. Yet, pure decentralized models may either require significant investment on the user side in terms of hardware and bandwidth, or may fail to provide the desired user experience. In this paper, we propose a middle-ground solution that mainly relies on a serverless *social overlay* in which links represent friendship relations, but occasionally complements it with inexpensive storage rented from the cloud. The experimental evaluation based on a real OSN dataset confirms that our solution provides the desired quality of service at a low cost.

Index Terms—Social Overlays, Decentralized Online Social Networks, Peer-to-Peer (P2P), Friend-to-Friend (F2F)

1 Introduction

Online Social Networks (OSNs) attract billions of users worldwide, and are today one of the most important Internet-based communication media. Yet, mainstream OSNs have been plagued by privacy issues from the start, due to their business model requiring users to surrender their private data to OSN providers. This model, albeit successful, has led to a number of important privacy incidents (e.g. [43]), motivating open, decentralized alternatives.

In this context, the key premise of decentralization is that it returns to the users control over their data, e.g., where to place it and with whom to share it. Initial, well-known efforts (e.g., [9]) relied on user-maintained servers to store and publish data. However, this requires users to either hire online hosting, which can be expensive, or to run their own servers at home, which can be cumbersome.

At the other end of the spectrum are serverless approaches (e.g. [4], [12], [22], [27]) that, inspired by peer-to-peer (P2P) computing, only require a computing device, installation of client software, and an Internet connection. The clients build an overlay network among nodes and collaboratively share their resources to maintain the system, e.g., replicating each other's data to increase availability, or donating bandwidth to route messages. These architectures free users from the burden of maintaining (and paying for) a server, but have issues of their own. First, most P2P systems are structured in a way that requires *strangers* to cooperate (e.g., pairs of randomly selected participants), leading to fundamental challenges in running these systems reliably in the absence of cooperation. Second, availability patterns in such systems are typically skewed [44], leading to further reliability, performance and quality-of-service issues [23].

This paper reconciles these two opposing approaches in Clops (Cloud-assisted profile dissemination over social overlays), a hybrid architecture for decentralized OSNs that neither requires costly centralized hosting services nor relies solely on a decentralized serverless network. Clops relies on two elements. First, a serverless fabric interconnects user devices into a *social overlay* (SO) whose links are directly determined by the friendship relationship: a social overlay link connects two nodes if and only if their owners are friends. The side effect of having an overlay network that *mirrors* the underlying social network is that identities and links become more difficult to forge. This, in turn, acts as a strong deterrent to malicious nodes as they can no longer freely penetrate the network, thus creating a more cooperative and secure environment [21], [32]. Indeed, the most widely used, censorship-resistant content sharing network today—Freenet [41]—relies on an SO for these very reasons.

Second, in Clops this decentralized communication layer works in conjunction with a cloud-based, centralized, highly-available, online storage service (e.g., Amazon's S3) which can store and serve encrypted objects quickly and reliably—and is *significantly less expensive* than full-fledged hosting. Clops exploits the complementary nature of these two elements via dedicated protocols that normally rely on social overlays for fast, decentralized, friend-to-friend communication, but occasionally resorts to cloud access to avoid the high delays caused by the skewed availability patterns of a purely decentralized solution.

Focus: Profile Dissemination over Social Overlays. Hereafter, we focus on the key goal of providing efficient *profile-based communication* among direct friends, which is the process through which users share content and post comments to their friend's profiles, viewing recent updates from friends as a newsfeed. Studies show [3] that profile-

- *Giuliano Mega is with SpazioDati s.r.l., Trento, Italy. E-mail: mega@spaziodati.eu*
- *Alberto Montresor and Gian Pietro Picco are with the Dept. of Information Engineering and Computer Science, University of Trento, Italy. E-mail: {name.surname}@unitn.it*
- *A very preliminary version of this paper appeared in [24].*

based communication among direct friends constitutes up to 80% of what users do on OSNs, making it the single most relevant feature an implementation must provide.

Clops enables decentralized, fast profile browsing by requiring nodes to keep local copies of their direct friends' profiles. These are usually small objects (few kilobytes) yielding an accrued size of only a few megabytes per friend. We review this and other system assumptions in Sec. 2.

Whenever a profile object is changed by the user, the update is proactively disseminated over the social overlay to keep the local profile copy on the friends' devices up-to-date therefore providing them with the information required to construct their newsfeeds locally.

Dissemination occurs only to direct friends over the social overlay. In other words, the updates posted to the profile of a user u are disseminated only over the *ego network* of u, i.e., the sub-network composed of u, her friends, and the interconnections among them. This choice brings two important benefits: *i)* it improves security, since the nodes in the ego network are friends with u, therefore less likely to act uncooperatively or maliciously towards u; *ii)* it induces a zero-spam dissemination strategy, as users that are not supposed to receive the post do not have to forward it.

The Problem: Fast and Efficient Update Dissemination. The main challenge in designing Clops is *how to disseminate profile updates quickly and efficiently over ego networks*, to approximate the performance of centralized OSNs in which profile updates happen almost instantaneously.

The problem is complex for two reasons. First, every serverless system must deal with *churn*—the process through which nodes enter and leave the network out of their own volition. We have shown in previous work [23] that, irrespective of the specific dissemination protocol, churn is a severe concern in social overlays. Unlike DHTs, easily repaired by picking one of the many available nodes, social overlays suffer from the limited set of links available for reconfiguration (i.e., only those among friends). This results in transient network partitions which incur large *delays* in update dissemination (in excess of 3 hours for 1% of the receivers)—unacceptable for a system of the scale of today's Facebook, and incompatible with our goals.

Second, social overlays inherit the peculiar structure of the social networks they mirror. These include, among others, the fact that the number of neighbors (node degree) changes dramatically across the network; a few high-degree nodes exhibit far more connections than average. In principle, this can be exploited to speed up dissemination. However, in practice *bandwidth requirements* must be taken into account, to avoid solutions that unrealistically overload nodes. We formally define the delay and bandwidth requirements of our dissemination problem in Sec. 3.

The Solution: A Hybrid Architecture. Since social overlays cannot deliver, on their own, the predictable performance we require, we must compromise w.r.t. decentralization. The main idea of Clops is to use the social overlay to disseminate updates quickly over the regions where it performs well, while relying on an inexpensive, highly-available cloud *storage* service (e.g., Amazon's S3) as an out-of-band channel to adaptively "patch" poorly-connected regions on-the-fly if and when necessary. While this brings centralization back in the loop, the key insight here is that services such as S3: *i)* are much cheaper than services like EC2 or other full-fledged hosting; *ii)* limit user exposure to the service provider via license agreements that promise data confidentiality and allow data to be encrypted—unlike centralized social networks, where information must be fully surrendered.

We describe our hybrid approach in Sec. 4. In a nutshell, every node u is associated to an always-available *profile store* in the cloud. Updates to u's profile are always written to the profile store, and then disseminated over the social overlay, making the profile store an always-available, always-current copy of u's profile. When a friend v 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 is not for free, 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) when it is not required. On the other hand, this requires a careful design of the *serverless dissemination layer* based on the social overlay, to reconcile the need for fast propagation of update and quench messages with the peculiar structure of social networks and the overall system bandwidth requirements. Clops uses gossip protocols as the main technique for both cases; we discuss protocol design and rationale in Sec. 5.

The experiments in Sec. 6 show quantitatively that Clops improves significantly over a purely serverless approach. To retain the ability to model and analyze our solutions we couple an availability model developed in the context of P2P [44]—which we expect, as we argue in Sec. 2, to be a good approximation of the real-world—with a real-world OSN dataset. We then show that, unlike the serverless approach, Clops is competitive with a centralized solution in terms of delays, while remaining efficient in terms of network usage and monetary costs, therefore confirming that its hybrid architecture strikes the right design balance for making decentralized OSNs an appealing reality.

Our design has significant elements of novelty w.r.t. related work. As discussed in Sec. 7, before our concluding remarks in Sec. 8, several works address profile availability and/or update dissemination. However, Clops is set apart by *i)* the choice of network topology (i.e., the social overlay) which brings about unique potential benefits, and *ii)* the combination of techniques we employ to overcome the limitations and challenges this choice of highly irregular network implies, specifically to promote high profile availability while enforcing, at low monetary cost, bounds on both latency and bandwidth usage.

2 System Model and Assumptions

Definitions and notation. We model a *social network* as an undirected graph $G = (V, E)$ where V contains users and E is the friendship relation among them. For each user $u \in V$, the function $f : V \to 2^V$ maps users to their set of friends, i.e. $f(u) = \{ v \mid (u, v) \in E \}$.

The *ego network* of a user u is the subgraph $G_u = (V_u, E_u)$, where $V_u = \{u\} \cup f(u)$ is given by u and her friends, while $E_u = \{ (v, w) \mid (v, w) \in E \wedge v, w \in V_u \}$ contains the interconnections among them. For convenience, we also define $f_v(u)$ as the set of neighbors node u has in G_v; i.e., $f_v(u) = V_v \cap f(u)$. We assume a node u has full knowledge of its ego network G_u, and is able to compute $f_u(v)$ without full knowledge of $f(v)$, as in centralized OSNs. We assume that users log from one device at a time, and thus we refer to users and nodes interchangeably. Finally, the *profile page* of a user u is denoted by $\mathbf{pp}(u)$.

Availability model. The serverless part of Clops is a network with $|V|$ participants, where each user u can be either logged in (online) or out (offline) at a given time instant.

In this context, we face two options for specifying online/offline behavior. The first one is to use publicly-available traces from measurement studies of P2P systems (e.g., [40]). Although real datasets are appealing, their use is challenging in our context, due to *i) size*: the social graphs required in our experiments are much larger than the networks in these traces, and it is unclear how to use the latter to simulate the former without biasing the results; *ii) duration*: even the longest traces are too short to accommodate a sufficient number of de-correlated experiments; *iii) noise*: most traces contain high rates of permanent departures (e.g., due to aliasing [40]), affecting statistical properties.

We therefore chose to use a well-known synthetic churn model by Yao et al. [44] that, being derived from the statistical behavior of measurement studies of P2P systems, offers a good approximation of real peer behavior while avoiding the problems above with traces. The Yao model associates an *alternating renewal process* to each node $u \in V$. In these processes, both session lengths (online time between a login and the next logout) and inter-session lengths (offline time between a logout and the next login) of u are given by random variables $\mathbf{X}^u_{on}$ and $\mathbf{X}^u_{off}$, drawn from node-specific distributions F^u_{on} and F^u_{off}. In this paper, we assume that both F^u_{on} and F^u_{off} are exponential distributions $\exp[\lambda]$ with parameter λ. We choose exponential distributions instead of heavy-tailed ones to make our simulations tractable [39].

Let $\mathbb{E}(\mathbf{X}^u_{on}) = \ell^u_{on}$ and $\mathbb{E}(\mathbf{X}^u_{off}) = \ell^u_{off}$. To model heterogeneous, skewed user availabilities, we draw the ℓ^u_{on} and ℓ^u_{off} for each node $u \in V$ from a pair of heavy-tailed, shifted Pareto distributions, as in Yao et al. [44]. These distributions are parameterized to yield global averages of $0.5/1$ hours for session/inter-session lengths, values taken from existing literature [35]. We then set the session length distribution for each node u as $F^u_{on} = \exp[1/\ell^u_{on}]$ and the inter-session one as $F^u_{off} = \exp[1/\ell^u_{off}]$. By drawing the *parameters* from heavy-tailed distributions, we can model skewed availabilities without sacrificing simulation tractability.

Several works in the literature (e.g. [13], [15]) argue in favor of using the distributions of (centralized) OSN session lengths as the basis for node availability instead. We disagree with such assessment as decentralized OSN clients are likely to be run in the background rather than turned on only when interacting with the OSN. Availability, therefore, is better approximated by file sharing sessions—the quintessential background program—than by OSN ones.

Local identities. We assume that each node $u \in V$ assigns local sequential identifiers to nodes in its ego network via a bijective function $lid_u : V_u \rightarrow \{0, 1, \cdots f(u)\}$ and, similarly, each $v \in V_u$ computes $lid_u(w)$ for all $w \in f_u(v)$. This is easily achieved if u recalculates local identifiers over G_u whenever $f(u)$ changes, and disseminates them as updates.

Realizing the social overlay. We assume that nodes are able to discover the IP addresses of their friends currently online to enable message exchange. The overlay management problem is outside the scope of this paper and can be addressed by leveraging existing decentralized server infrastructures such as Jabber/XMPP [34]. NATs and firewalls are assumed to be dealt with by existing techniques [33].

Authentication, access control, and privacy. We assume that each node u is identified by a pair of asymmetric keys $(Priv_u, Pub_u)$. Most of the security advantages social overlays bear come from the fact that public keys are assumed to be exchanged directly by users, out-of-band. In this setting, forging identities is difficult, hence Sybil attacks [10] are hard to mount. Further, the limited knowledge of network structure implies that disrupting it (e.g., with Eclipse attacks [38]) is hard too, as it requires compromising a large number of private keys. Finally, once keys are established, guaranteeing authenticity is a matter of signing every update o generated by u with $Priv_u$. More complex access control mechanisms are possible [12], but outside of the scope of this paper.

Update frequency and size. Although users share content of different nature, most of what is shared in the profiles of modern OSNs are small objects under $150\ kB$. This is in line with figures observed in other works in the literature [15]. Further, we assume updates to be *sparse*, i.e. most users post from less than one to few updates a day [15], [30]. Therefore, *concerns about latency take priority over bandwidth* when gauging the quality of a dissemination technique.

These profile objects include text snippets, messages, and pictures. While users also share larger objects, e.g., videos or high-resolution pictures, these are not necessarily part of profile pages, as they are typically *linked* from third-party services such as YouTube. This does not imply loss of privacy, however, as the *metadata* on who shared the video with whom, who liked it, and any comments made to it remain safely confined to our system.

Cloud access. We assume the existence of a highly available, cloud-based service which nodes can access to store and retrieve data. This cloud service allows users to create personal storage areas under their control, i.e, they can selectively grant access to other users.

User profiles are small (around 10MB [15]) and likely within the free quota currently allowed by cloud providers. 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 [2], where users accessing data are charged for it. In other words, if a user v decides to download the new updates from a friend directly from u, it is up to v to pay for the download costs. This 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 two parts:

- *Storage costs.* Storing data on the cloud costs less than $3¢$ per GB per month at time of writing. As profiles are at

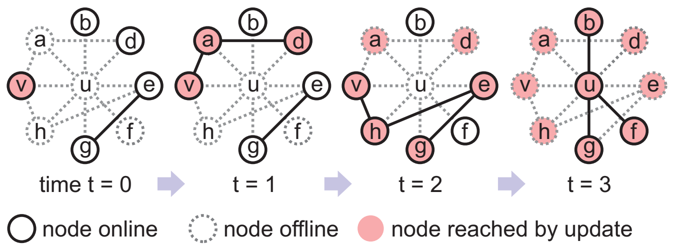


Fig. 1. The interplay between node availability and dissemination delay.

most a few MBs, we do not consider their storage costs.

- *Cost per request.* In S3, a user pays $0.4¢$ for every $10,000$ GETs and $0.5¢$ every $1,000$ PUTs. We assume a similar cost model where read requests are cheaper than writes.

3 Problem: Fast and Efficient Update Dissemination on Social Overlays

In an OSN, updates to a profile page $\mathbf{pp}(u)$ are generated either by u (e.g., when posting a status update) or its friends (e.g., when they add content or comments to u's "timeline"). In Clops, nodes keep local copies of their friends' profile pages; therefore, proper mechanisms are required to proactively disseminate updates to u's profile to all copy holders (*receivers*), i.e., to all nodes $v \in f(u)$.

This *update dissemination problem* is at the core of the work presented here and, in general, it is fundamental to any decentralized OSN design. In our case, the choice of relying on the social overlay as the main vehicle for dissemination brings advantages (as discussed in Sec. 1) but also complicates the problem due two reasons, discussed next: the presence of *churn*, which causes unpredictable communication delays, and the *structural hurdles* posed by the peculiar nature of social networks, which bears an impact on bandwidth requirements.

3.1 Churn: Communication Delay Requirements

Friends who wish to communicate are often not online at the same time. This means that updates may have to be relayed across a chain of intermediate nodes before reaching their destination, and this may lead to delays.

The dynamics through which such delays arise is depicted in Fig. 1 where, at time instant $t = 0$, node v starts disseminating an update over an egonet G_u. Since nodes a, u, and h are offline, the update initially cannot 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.

In practice, social networks (and hence social overlays) are rarely as regular as in Fig. 1; irregular clustering often yields topologies as in Fig. 2. In the presence of churn, these topologies may easily lead to partitions, as in the case where node b in Fig. 2 is temporarily offline, preventing communication from the sub-networks v and w belong to. These situations occur quite frequently, and are the biggest hurdle to the use of social overlays in decentralized OSNs.

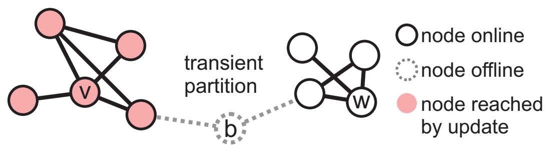


Fig. 2. Transient partitions in social overlays.

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

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 node that logs in infrequently does not care if an update was posted long ago, 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 $\mathbf{rd}(v, w, t_0)$.

We have demonstrated in [23] that, under any purely P2P solution, both $\mathbf{ed}$ and $\mathbf{rd}$ can become very 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—unacceptable for modern OSNs.

Putting a Bound on Delays. Given the requirement for fast dissemination, our goal is to solve the update dissemination problem while providing an acceptable user experience. Formally, we can express this goal as a *target delay bound* δ on update delivery:

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

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. E.g., suppose we set $\delta = 20$ minutes and a receiver logs in 5 minutes per day. From the point of view of Eq. (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 unnecessarily large. We need a stronger bound based on $\mathbf{ed}$. Yet, bounding $\mathbf{ed}$ directly is not possible: 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 $\mathbf{ed}$. 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 $\mathbf{R}_{\mathit{off}}$ of w until its next login. Formally:

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

The slack time is a random variable, whose probability distribution results directly from the availability model. The bound in Eq. (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 honoring Eq. (2) we are also automatically honoring Eq. (1).

3.2 Structural Hurdles: Bandwidth Requirements

Social networks have peculiar structural properties, e.g., skewed degree distributions and irregular clustering [22]. In Clops, the social network is directly *mirrored* into the social overlay used for dissemination, which therefore inherits these structural properties. Consequently, high-degree nodes tend to sit on more dissemination paths than low-degree ones [18], and in principle must perform more work.

Fig. 1 illustrates the issue: u is the highest-degree node in the network, and we could disseminate any update over the graph by simply asking u to do it. Yet, this means u must provide enough bandwidth, which may be unreasonable depending on how large u's degree is and how often new updates are generated. In general, we assume u has a limited *bandwidth budget* that our protocols must respect; we discuss and formalize this issue in Sec. 5.3, where we illustrate a protocol capable of honoring this bandwidth requirement.

4 Solution: Cloud-enhanced Dissemination on the Social Overlay

Social overlays cannot meet *by themselves* the delay and bandwidth requirements for all nodes, as discussed in the previous section. Therefore, in this paper we propose a hybrid solution that, by combining a decentralized social overlay with occasional cloud access, is able to reconcile the two seemingly opposing goals of respecting *delay bounds* and *bandwidth budgets* in the face of skewed availability patterns, while incurring modest monetary costs.

Once the leap to cloud access is made, it would seem that simple solutions exist. For instance, a simple way to "patch" partitions like those in Fig. 2 would be to associate each node u to an *alias* $\tilde{u}$ in the cloud, activated on-demand to satisfy requests on behalf of u when u is offline. These cloud aliases would effectively remove transient partitions and their associated delays. This solution has, however, a number of drawbacks. First, running an alias on the cloud (e.g., EC2 [2]) for extended periods of time is too expensive. Second, as Shakimov [36] points out, activating an alias on-demand requires specialized (and hitherto non-existing) support from the cloud provider, e.g., to constantly monitor the state of the alias and take over should it crash. Finally and most importantly, it provides the cloud provider with full access to the state of the running software, potentially allowing access to sensitive information such as private keys, and violating our principle of minimum exposure.

4.1 Key Insight: Cloud-based Profile Stores

The problems above led us to an alternative solution where $\tilde{u}$ is not a full clone of u, rather a simple high-availability *profile store* in which $\mathbf{pp}(u)$ is kept.

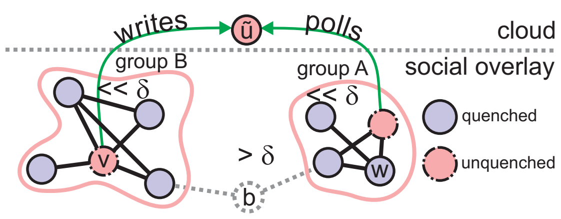


Fig. 3. Delay groups and Hybrid.

If v wants to post something to $\mathbf{pp}(u)$, it simply writes this update directly to $\tilde{u}$. Cloud services such as S3 [2] provide means to ensure that only authorized users are allowed to write to $\tilde{u}$. By adopting the requester-pays model of S3, we ensure that users have complete control over costs. 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 OSNs 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 *polling*. We describe two solutions, evaluated in Sec. 6, that strike different trade-offs between complexity and performance.

A naïve solution: PurePoll. Our baseline protocol, PurePoll, works by having each node $w \in f(u)$ independently poll $\tilde{u}$ every δ time units, retrieving any updates posted in the meantime. If w happens to be offline after δ time units have passed, then w accesses the cloud immediately once it logs back in. This approach allows us to circumvent the transient partitions of Fig. 2 through an out-of-band channel, satisfying Eq. (2).

Exploiting the social overlay: Hybrid. 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 Fig. 2—we would be left with a set of disjoint groups for which the *internal* delays are low, as illustrated in Fig. 3. We call these *delay groups*. To disseminate a message over a set of delay groups while respecting the delay bound δ, we need to ensure that *at least one* node in each of these groups actually accesses the cloud every δ time units. The other nodes can then get the update from this node over the social overlay and avoid accessing the cloud themselves.

Our second approach, named Hybrid, does just that. Each node $w \in f(u)$ keeps track of the last time it heard from u. Whenever w goes for more than δ time instants without hearing from u, it polls u's profile store to check for new updates. If there are any, w downloads them from the cloud and disseminates them over the social overlay by means of an appropriate *dissemination protocol*, described in Sec. 5. Otherwise, w disseminates a special `quench` message that contains the time t_0 at which w accessed the cloud and found nothing new. This message informs other nodes that they can refrain themselves from accessing the cloud for an extra δ time instants. We call this *access quenching*.

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. Further, note that delay groups are exploited *implicitly* and do not have to be constructed nor maintained—nodes that are part of different delay groups will simply never be able to quench one another as messages will take too long to propagate over the social overlay.

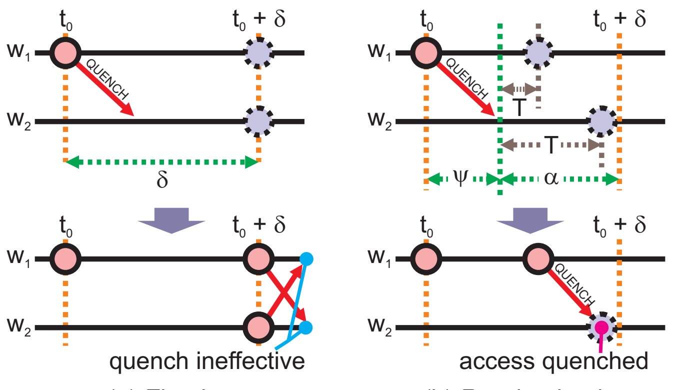
Randomizing cloud accesses. A side effect of the `quench` mechanism is that it induces nodes belonging to the same delay group to synchronize their accesses to the profile store, as illustrated in Fig. 4a: node w_1 initially accesses $\tilde{u}$ at time t_0 and, having found no new updates, schedules its next access to $t_0 + \delta$ to respect the soft delay bound. At the same time, w_1 propagates this knowledge over the social overlay through a `quench` message. Upon receiving such message, node w_2 schedules its next access to $t_0 + \delta$ as well. At time $t_0 + \delta$ (Fig. 4a), both nodes access $\tilde{u}$. Having again found no updates, they disseminate their `quench` messages, but to no effect since the accesses are too close to one another.

We address the problem by scattering access times near $t_0 + \delta$. When accessing the cloud or when receiving a `quench` message, nodes set their access time to $t_0 + \psi + \alpha$, where ψ is a fixed component large enough to make quenching effective (Fig. 4b), and α a uniformly random value between 0 and $\delta - \psi$, satisfying the soft bound in Eq. (2).

4.2 Solution Details

In HYBRID, every node $w \in f(u)$ keeps track of three variables: *i)* the *last seen* timestamp *last*$[u]$ representing the last time at which w has heard any news from u *ii)* the version number *version*$[u]$, used to determine whether its local version of $\mathbf{pp}(u)$ is up-to-date w.r.t. the profile store $\tilde{u}$ *iii)* its current, randomized target delay bound *target*$[u]$.

Whenever the target delay bound is crossed without hearing any news from u, w accesses the cloud, as shown in Alg. 1. The first actions w takes, since it is about to get the latest updates concerning u, is to suspend the dissemination



(a) Fixed. (b) Randomized.

Fig. 4. Access quenching.

Algorithm 1: OnTimeout

```
Trigger: Target delay bound is crossed
    (clock() − last[u] > target[u]).
1  $M \leftarrow SO$.query(⟨quench, $u, \cdot$⟩)
2  foreach $m' \in M$ do $SO$.remove($m'$)
3  last[$u$] ← clock()
4  $U \leftarrow$ cloud.list($u$, version[$u$])
5  if $U \neq \emptyset$ then
6      version[$u$] ← max($U$)
7      foreach $id \in U$ − delivered do
8          upd ← cloud.download($u$, id)
9          deliver(upd)
10         delivered ← delivered ∪ {id}
11         $SO$.disseminate(⟨update, $u$, last[$u$], id, upd⟩)
12 else
13     $SO$.disseminate(⟨quench, $u$, last[$u$], version[$u$]⟩)
14 target[$u$] ← $\psi$ + uniform(0, $\delta - \psi$)
```

Algorithm 2: OnReceive

```
Trigger: Message $m$ is received from the social overlay.
1  switch $m$.type do
2      case update
3          if $m$.id ∉ delivered then
4              deliver($m$.upd)
5      case quench
6          if $m.t \leq$ last[$u$] or clock() − $m.t > \delta$ or
7            $m$.version ≠ version[$u$] then
8              $SO$.remove($m$)
9  if $m.t >$ last[$u$] and version[$u$] = $m$.version then
10     last[$u$] ← $m.t$
11     $M \leftarrow SO$.query(⟨quench, $u, \cdot$⟩)
12     foreach $m' \in M : m'.t < m.t$ do $SO$.remove($m'$)
13     target[$u$] ← $\psi$ + uniform(0, $\delta - \psi$)
```

of any `quench` messages over the social overlay (SO) and to update *last*$[u]$ (lines 1-3).

Then, w downloads into the list U all the identifiers of u's updates that are more recent than *version*$[u]$ (line 4). If U is non-empty (line 5), *version*$[u]$ is set to the largest update version number in U, and all the updates not yet received are downloaded from the cloud. The precise nature of the cloud operations depend on the API provided. For example, in S3 [2] this could be obtained through the versioning mechanism and the "GET bucket object versions" primitive.

After download, w delivers the updates to the application layer (lines 9-10) and to the serverless dissemination layer (line 11) described in Sec. 5, from where they are spread over the ego network G_u. As shown in line 11, update messages contain four fields besides the message type descriptor: the identifier of the owner of the profile page to which the update is addressed (u), the timestamp of when the update was last downloaded (*last*$[u]$), the identifier of the update (*id*), and the update itself (*upd*).

If instead no updates are found (line 12), w disseminates a `quench` message with the identifier of the profile page owner, the access timestamp, and the latest version of $\mathbf{pp}(u)$ it knows, *version*$[u]$ (lines 12-13). Including *version*$[u]$ is important as a `quench` message indicates the absence of updates in reference to the latest version of $\mathbf{pp}(u)$ known by w, and different nodes might be stuck with different versions. Finally, w randomizes its target delay bound *target*$[u]$, as described in Sec. 4.1.

Upon receiving a message m, a node w executes Alg. 2. If m is an `update` message (line 2) whose content is unknown to w, it is delivered to the application layer (line 4). This will implicitly cause *version*$[u]$ to be updated in case w has delivered all updates that precede m (not shown). If m is a `quench` message and its timestamp is older than *last*$[u]$, or the difference between the current local time and m's timestamp is larger than δ, then the message is too old and w does not disseminate it (line 8). The same happens if the version number in the message does not match *version*$[u]$, as this means that the `quench` is either too old (if $m.version <$ *version*$[u]$), or too new. The latter may happen when, e.g., w spends time offline and, upon coming back, receives a `quench` from a node v who was online for the whole period and, having accessed the cloud recently, found no updates.

Regardless of the message received, w updates *last*$[u]$ and its target delay bound—which will cause quenching at w—whenever the timestamp of the message is more recent than its own, and its version number is the same as *version*$[u]$. The latter constraint prevents w from quenching access in response to messages from nodes with newer versions of the profile page $\mathbf{pp}(u)$. In case the constraints hold, w also stops disseminating any `quench` messages with timestamp smaller than that of m (lines 11-12), 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 Alg. 1. Yet, there could be some neighbor of w in the social overlay with recent updates on u, which would make cloud access unnecessary. To reduce the chances of such an event happening, we allow joining nodes a short grace period on login (5 seconds, in our current implementation), thus giving neighbors enough time to send any new information.

5 The Serverless Dissemination Layer

The serverless dissemination layer operating on the social overlay is a critical component of Hybrid's overall performance. Its role is to disseminate `update` (or `quench`) messages *as quickly as possible* to nodes within low-latency paths from a sender. The intuition has already been given in Fig. 3: once a member of a delay group accesses the cloud, we want information to spread quickly to other members to quench accesses or promote fast updating.

Since *speed* is the primary concern, we choose to build our dissemination layer entirely on top of fast, *push gossip protocols* [8]. Apart from being fast, these protocols are known for their simplicity, scalability, and churn resilience. Push gossip protocols share a common structure: they all work in periodic "rounds" of length t_r which are *not* synchronized among participants. At each round, whenever there is an update to be disseminated by a node v, the protocol at v will:

1) randomly select a neighbor u;
2) push the update to u;
3) decide whether to run another round in t_r time units or instead *halt*, i.e., stop the protocol locally.

To disseminate multiple updates simultaneously, v can run multiple instances of this simple base protocol in parallel.

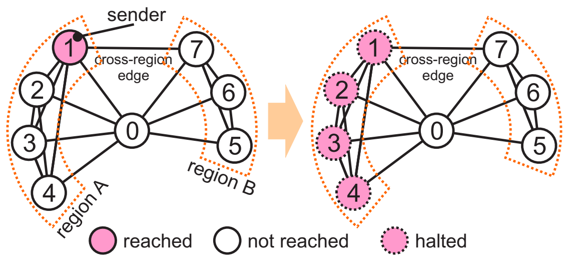


Fig. 5. Irregular clustering and Demers.

Since push protocols must be fast, the round length t_r for a push protocol is usually small; hereafter, we assume w.l.o.g. $t_r = 1$ second.

5.1 The Need for New Gossip Protocols

Ideally, we would like to reuse existing gossip protocols. Unfortunately, we found [22] that these cannot be applied in our context for the same reason most protocols fail in social overlays—common assumptions on network structure fail to hold. We shed more light into this aspect by examining the issues faced when applying the classic Demers' rumor mongering protocol [8] to our problem. Demers instantiates the generic gossip protocol template given above by selecting neighbors uniformly at random and by halting update dissemination with probability p if the neighbor has already received the update (feedback-coin [8]).

Clustering. If the network is clustered approximately uniformly, Demers guarantees that the update will spread to most of the nodes before duplicates appear and nodes begin to halt. In ego networks, however, nodes are clustered into regions that are well-connected internally (Fig. 5) but not to each other (i.e., forming local *communities* [17]). This biases random selection, causing it to occur more often within a region than outside it, leading to *i)* the fast spreading of updates within regions, and *ii)* the early appearance of duplicates, causing the process to halt prematurely (Fig. 5). This effect is analyzed experimentally in [22], showing that up to 20% of receivers do not get the update, even with a low halting probability p and no churn.

Load and node degrees. In push gossip, the frequency with which a node gets selected by its neighbors (and, therefore, receives a message) depends on its in-degree: the higher the degree, the larger amount of load the node will have to endure. For update traffic, this is not such an important issue, as it is rather sparse, with reported averages ranging from less than one, to a few posts per day [30]. On the other hand, `quench` messages must be disseminated frequently, and can easily lead to problems.

Our Requirements. In a nutshell, we want to design efficient push gossip protocols that must: *i)* work efficiently under non-uniform clustering conditions; *ii)* cope with two distinct traffic classes—`update` and `quench` messages—without sacrificing speed and without overloading high degree nodes, as discussed in Sec. 3.

These requirements lead us to two push gossip protocols, described next, which are an important contribution of our work: Quick and Thrifty. Quick is geared towards sparse `update` traffic and ensures that updates get delivered quickly under widely varying clustering conditions.

THRIFTY, instead, is geared towards `quench` messages and ensures that these get delivered fast enough, while providing guarantees on the loads incurred in high degree nodes.

5.2 QUICK Updating

Recall that Demers assumption on uniform clustering causes it to stop disseminating too quickly. One way to avoid the problem would be by *flooding* instead—i.e. a node does not stop disseminating an update until it sends it to each of its neighbors at least once. But this generates too much traffic, which we remedy with a modification to flooding that we name QUICK. In a nutshell, QUICK nodes flood, but they also piggyback with each message a history containing the nodes in the ego network known to have already received the update, and avoid selecting them again.

Message histories. Formally, let o be an update, and let $K_{v,o} \subseteq V_u$ contain the intended destinations of o known by v to have received o. Let $E_{v,o} = (V_u \cap V_v) - K_{v,o}$ denote the common friends of v and u (including u) *eligible* for selection at v when considering o. Then, at each round, node v:

1) selects $w \in E_{v,o}$ uniformly at random;
2) sets $K_{v,o}$ to $K_{v,o} \cup \{w\}$
3) sends a message $\langle o, K_{v,o} \rangle$ to w, where the *message history* of o known by v is piggybacked with o;
4) when v receives a message $\langle o, K_{z,o} \rangle$ from z, it sets $K_{v,o}$ to $K_{v,o} \cup K_{z,o}$;
5) if $E_{v,o} = \emptyset$, v stops disseminating. Otherwise v schedules the next round.

Since the social network changes slowly and each node has a unique local index into the ego network (the lid_v function, Sec. 2), we can implement message histories as simple bit arrays. For example, in Fig. 5 we would piggyback with each message an array with 10 bits. If bit k is set, node k already received the message. For large ego networks, the array can be compressed for bandwidth efficiency [14].

QUICK reduces traffic by avoiding duplicates. Less obviously, it also boosts dissemination speed: nodes in a clustered region typically enter histories early, causing links pointing outside the region (e.g., the cross-region edge in Fig. 5) to be activated earlier, thus rendering dissemination more "parallel" and, thus, faster. These effects, as well as QUICK's message overhead, were studied in detail in [22].

Timeouts. Since we want QUICK to spread over delay groups without causing too much overhead, dissemination time is limited by a timeout parameter t_{out}; if a node v goes for more than t_{out} time units without finding a new neighbor to push to, then dissemination spontaneously halts. Hereafter we assume $t_{out} = 2$ minutes, which we verified to be enough to ensure good coverage.

5.3 THRIFTY Quenching

Nodes in our system participate in several ego networks, and may have to help disseminate over several of them concurrently. Fig. 6 shows an example, where node u must disseminate both over G_u (his own ego network) and G_v. An immediate consequence is that running QUICK with a round length of $t_r = 1$ s as proposed earlier would only be doable if we assume that u must not disseminate over too many ego networks at the same time—i.e., if we assume that

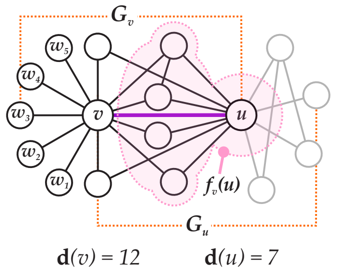


Fig. 6. Two ego networks sharing nodes.

posts from friends are somewhat desynchronized in time. This is a reasonable assumption for update traffic, as it is sparse. What would happen, however, if all of u's friends (e.g., all thousand of them) decided to post an update at exactly the same time? Node u would have to send as much as a thousand messages per round, a hefty bandwidth tag.

But this is precisely the effect that the mechanism we propose has on the network: new `quench` messages are posted continuously and in a succession by members of delay groups (who are normally neighbors), and cause higher degree nodes to be overloaded. This problem is solved by THRIFTY, a protocol that provides enough speed for `quench` messages and is aware that all nodes—even high degree ones—may be constrained by a *bandwidth budget*.

In a nutshell, THRIFTY is a gossip protocol that allows the dissemination process to respect probabilistic bandwidth budgets despite extreme differences in node degrees and the fact that *actual* node degrees vary due to churn. It achieves these goals by *re-weighing selection probabilities* taking into account both *i)* the differences in degrees between selecting and selected nodes, and *ii)* network dynamism, by means of network parameters that can be easily estimated locally.

5.3.1 Modeling Bandwidth Requirements

Before describing our solution, we formalize the bandwidth requirements we stated informally in Sec. 3.2.

For a given node v, we define its *bandwidth budget* as a pair of mappings (o_v, i_v) such that, for each $u \in f(v)$, $o_v(u)$ and $i_v(u)$ represent the *average number of messages-per-second* v is willing to send and receive, respectively, when helping disseminate updates over G_u. In other words, $o_v(u)$ and $i_v(u)$ represent how much v is willing to help disseminate updates posted to $\mathbf{pp}(u)$ by u and her friends.

To simplify our analysis, we assume that the dissemination layer is always *saturated*, i.e., that the push protocols never have an opportunity to terminate because nodes "post" new `quench` information faster than the termination condition can be met. This means that *all nodes send at every round*. As discussed later, this is an accurate representation of what happens with quenches under HYBRID.

Outbound bandwidth. Consider node v and the outbound bandwidth $o_v(u)$ it budgets for $u \in f(v)$. Let $\mathbf{O}_v(u)$ be the random variable representing the number of messages that v sends towards G_u at some gossip round. Recalling that $V_u = f(u) \cup \{u\}$ and a node sends at most one message per round towards an ego network, at round r we have that:

$$\mathbf{O}_v(u) = \begin{cases} 1, & \text{if } v \text{ sends a message to some } w \in V_u; \\ 0, & \text{otherwise.} \end{cases}$$

The requirement is that the actual outbound traffic respects the budget, i.e., $\mathbb{E}(\mathbf{O}_v(u)) = o_v(u)$.

Inbound bandwidth. Consider node v and the inbound bandwidth $i_v(u)$ budgets for the ego network G_u, $u \in V_v$. For an arbitrary round of our gossip protocol, let $\mathbf{C}_w$ be a random variable such that, at some round r:

$$\mathbf{C}_w = \begin{cases} 1, & \text{if } v \text{ is contacted by } w \in V_u; \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Further, let $\mathbf{I}_v(u)$ be a random variable representing the number of contacts v receives from nodes in G_u in a gossip round. Clearly, the contacting nodes in G_u have to be both in $f(v)$ and V_u; i.e. they have to be in $f_u(v)$ (defined in Sec. 2). $\mathbf{I}_v(u)$ can then be expressed as:

$$\mathbf{I}_v(u) = \sum\nolimits_{w \in f_u(v)} \mathbf{C}_w \tag{4}$$

$\mathbf{I}_v(u)$ represents the actual inbound bandwidth v dedicates to dissemination over G_u at any given round, and our primary goal becomes ensuring that $\mathbb{E}(\mathbf{I}_v(u)) \leq i_v(u)$—i.e., that usage respects the budget.

5.3.2 Meeting Bandwidth Requirements

Outbound bandwidth. Building a protocol that respects the outbound bandwidth budget is simple, as the number of messages v sends depends only on local actions made by v itself. To ensure that v sends, on average, $o_v(u)$ messages per second towards G_u, v must skip a round with probability $1 - o_v(u)$. Since we are working with a gossip round length $t_r = 1$ s, we have that $o_v(u) \leq 1$, i.e., v sends at most 1 message/s *per ego network* it participates in; adapting to other round lengths would be trivial. This leads to:

$$\mathbb{E}(\mathbf{O}_v(u)) = 1 \cdot o_v(u) + 0 \cdot (1 - o_v(u)) = o_v(u)$$

While we could have simply had v send a message every $1/o_v(u)$ rounds towards G_u, our scheme ensures that sends from different instances of the push protocol synchronize with low probability. Indeed, the total output bandwidth consumed by v over all ego networks it participates is, at a given round, given by:

$$\mathbf{O} = \sum\nolimits_{u \in f(v)} \mathbf{O}_v(u)$$

which follows a *Poisson binomial distribution* [42] with parameters $\{\, o_v(u) : u \in f(v) \,\}$ and tends to have low variance. Clearly, v might always opt to skip rounds if the actual available bandwidth is not enough, but the protocol itself ensures that usage remains acceptable and predictable.

Inbound bandwidth. Controlling the inbound bandwidth needed by node v is more challenging, as it depends directly on how frequently *v is contacted by other nodes* at each round.

The intuition for THRIFTY is that nodes can avoid overselecting high degree neighbors by "vetoing" selection with a probability that is proportional to the neighbor's degree and bandwidth budget. We develop this intuition by looking at the simple network in Fig. 6, and considering v's budget $i_v(v)$ towards its own ego network, which we assume to be $i_v(v) = 1$ for now; i.e., that v wants to be contacted, on average, once per round by its neighbors in G_v.

For every node $u \in f(v)$, denote their degree in G_v as $deg(u) = |f_v(u)|$. A way to achieve fairness while honoring our average bandwidth constraint would be by making it so that every neighbor of v selects it with probability $1/deg(v)$ at every round—i.e., every neighbor gets the same chance. Achieving this goal for nodes $w_1 \dots w_5$ in Fig. 6 is simple—have them select v with probability $1/deg(v)$, or sit idle with probability $1 - 1/deg(v)$. Clearly, we could achieve our goal by having u follow the same strategy, but this is suboptimal—u has other neighbors in G_v, and it should select those when it cannot select v. Further, we also want u to be *fair* in its neighbor selection—i.e., we want them to be selected with equal probability. Finally, we want to deal with cases in which $i_v(v)$ is an arbitrary number, not just 1.

To reconcile all these goals, we alter the selection procedure at u as follows. At each round, node u:

1) selects a neighbor $w \in f_v(u)$ uniformly at random;
2) draws a sample α from $\mathsf{uniform}[0,1]$ and:
   a) if $\alpha < \min(1, deg(u)/deg(w)) \times i_w(v)$ selects w;
   b) skips a round, otherwise.

We refer to the expression $\min(1, deg(u)/deg(w)) \times i_w(v)$ as the *reweighing factor*. The critical property of the reweighing factor is that it is both directly proportional to u's degree (high-degree nodes are less likely to skip rounds) and inversely proportional to w's degree (high-degree neighbors are more likely to be skipped). More formally:

Theorem 1. THRIFTY *respects the inbound bandwidth $i_w(v)$ for all $w \in V_v$.*

with a proof provided in the supplemental material [25].

THRIFTY relies on the assumption that u *knows* $i_w(v)$, otherwise it cannot compute the right reweighing factor for it. However, since w and u are friends, w can easily communicate its bandwidth budget $i_w(v)$ to u by piggybacking the information on update traffic.

5.3.3 Dealing with Clustering and Churn

Clearly, we want THRIFTY to perform well under nonuniform clustering conditions. We therefore adopt the same mechanism we used for QUICK in Sec. 5.2—node u piggybacks a message history with every `quench` message o it sends around. Since u is only allowed to select from $E_{u,o}$, however, we have to change the reweighing factor from $\frac{deg(u)}{deg(w)}$ to $\frac{|E_{u,o}|}{deg(w)}$. We show in [25] that this does not affect the ability to correctly meet the bandwidth budget.

Moreover, we have implicitly assumed thus far that network degrees remain static over time. Churn, however, implies that they change, often significantly. Fortunately, for THRIFTY to work, all u needs to know about a neighbor w is the *average* number of neighbors that w sees online over G_v, referred to as $adeg(w)$. $adeg(w)$ can be estimated firsthand by w, who keeps track of how many neighbors it sees online. Node w can then disseminate its estimate of $adeg(w)$ over G_v a few times a day—every 6 hours, in our case. Once a node disseminates its average, it resets its estimate to $deg(w)$ and starts counting again. Therefore, nodes that do not spend enough time online, will have pessimistic estimates, which will cause their budgets to be underutilized and therefore respected. Averages are disseminated using THRIFTY together with `quench` traffic. As our evaluation shows, using $adeg(w)$ instead of $deg(w)$ in the reweighing factor is sufficient to respect the budget.

Algorithm 3: THRIFTY

```
1 if $\mathcal{U}[0,1] \geq o_u(v)$ then
2     return null;            % Skips to respect $o_u(v)$
3 $w \leftarrow \mathsf{random}(E_{u,o})$
4 if $\mathcal{U}[0,1] < \mathsf{min}(1, |E_{u,o}| / adeg(w)) \times i_w(v)$ then
5     return $w$
6 return null;                % Skips to respect $i_w(v)$
```

In practice, such processes are cyclostationary due to time-of-day effects [40]; a moving average disseminated periodically would probably be a better fit. An analysis of these issues is, however, outside the scope of this paper.

5.3.4 *Protocol*

The resulting protocol run by u to disseminate updates over G_v is shown in Alg. 3. It starts by skipping a round (returning null) with probability $o_u(v)$ (lines 1-2), to respect the outbound bandwidth budget $o_u(v)$. Then, it selects a neighbor from $E_{u,o}$ uniformly at random (line 3) and reweighs the selection probability according to the rules we outlined (line 4). This causes either w to be selected or the current round to be skipped.

6 EVALUATION

In this section we evaluate our solution towards two goals: 1) provide supporting evidence that it performs significantly better than a pure serverless approach, while being competitive with centralized approaches; 2) estimate what kind of monetary and network costs one should expect from running it. Additional details about the dataset we use, the rationale for the experimental setup, and the chosen metrics, are available in the supplemental material [25].

6.1 Baselines and Experimental Setting

We compare HYBRID against four baselines: *i)* PUREPOLL, the naïve protocol described in Sec. 4.1; *ii)* LAVISH, a variant of HYBRID that disseminates `quench` messages using QUICK instead of THRIFTY and allows us to quantify the benefits of the latter; *iii)* PUREP2P, which relies solely on the social overlay and allows us to quantify the benefits of our cloud-assisted strategy; and *iv)* SERVER, emulating a centralized approach akin to Facebook.

Dataset. Protocols are evaluated over a sample of 100k ego networks picked uniformly at random from the Orkut crawl in the Stanford Large Network Dataset Collection (SNAP) [19]. Being the largest (Facebook-like) OSN crawl on SNAP, the dataset allows us to select a larger number of non-overlapping ego networks, thus maximizing structural variability and ensuring the relevance of our numbers.

The graph contains 3 million vertices (i.e., 3M ego networks), 117M edges, and has an average clustering coefficient of 0.171. Our sample covers 2.1M unique vertices, has around 39M unique edges, and similar clustering coefficient.

Simulation approach. For each ego network G_u in our sample, we pick *one* source node $v \in V_u$ uniformly at random. This leads to the set of 100 000 (*source*, *ego network*) pairs we use for simulations. For each pair (v, G_u) in our sample, we perform the following experiment. 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 the steady-state regime, we run the simulation for a *burn-in period* γ in which no measurement is taken: we simulate the churn model and, in the case of PUREPOLL and HYBRID, we also simulate the polling of the cloud and the `quench` messages.

The goal is to obtain a representative online/offline network configuration before we start injecting updates, but also a representative `quench` traffic, and a representative "phase shift" for the last seen timestamps *last*$[u]$ at the various nodes. We have empirically established, with heuristics similar to [1], that setting $\gamma = 48$ hours is enough to mitigate our initial transient. To derive the estimators (e.g., sample averages) we require, we run two types of simulations:

- *Delay simulations* where we simulate updates originating at v, and measure delays towards receivers in $V_u/\{v\}$. The experiment progresses by waiting for the first login of the source v, at which point we cause v to post an update to $\mathbf{pp}(u)$. We assume w.l.o.g. that nodes always post their updates at the start of one of their sessions (since the system is assumed to be running in stable state, it makes no difference at which point of the session the node posts). We then wait for the update to reach each of the nodes in $V_u/\{v\}$, and measure the delays. Experiments are repeated 100 times.
- *Cost simulations* where no updates are simulated, but nodes in V_u are continuously accessing the cloud and disseminating `quench` messages, allowing us to estimate the stable-state costs of the protocol. We let the experiment run for another 800 simulated hours without any updates. During this period, we measure both cloud access frequencies and network usage. This experiment is only run once.

These simulations are performed separately because delay simulations are irregular in duration, and introduce a bias if cost metrics are calculated over them.

6.2 Metrics

This provides a high-level description of the metrics we calculate; more details are available in [25].

Delay. With reference to Sec. 3.1, for each source destination pair (u, v) we compute their *average* end-to-end and receiver delays, which we refer to as $\mathbf{aed}(u, v)$ and $\mathbf{ard}(u, v)$.

Monetary costs. We estimate $\mathbf{ycs}(w, u)$, the yearly costs incurred by node w for keeping up-to-date with the updates of one profile page $\mathbf{pp}(u)$. This is obtained by computing, at each node $w \in V_u$, the average number of accesses $\mathbf{acs}(w, u)$ that w incurs on the cloud *per hour* over the duration of the cost experiment ran over G_u, aggregating them over the period of one year, and multiplying by ρ, the cost of a GET request. We then estimate the overall yearly spendings as

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

by accounting for the ego networks w participates in, assuming costs increase linearly w.r.t. node degree.

Network costs. Network usage is assessed by measuring the average bandwidth consumption at each node, in terms of message/s processed. Similarly to what we did for cloud accesses, let $\mathbf{amsg}(w, u)$ represent the number of messages we estimate w to process per second during the cost simulation ran over ego network G_u. As before, this gives us a rough figure for the costs incurred by w while keeping up-to-date with $\mathbf{pp}(u)$. We then adopt the same approximation model as in $\widehat{\mathbf{cs}}$ when computing the total message processing rate of $\widehat{\mathbf{msg}}(v)$ at v, and multiply $\mathbf{amsg}$ by $|f(w)|$:

$$\widehat{\mathbf{msg}}(w) \sim \mathbf{amsg}(w, u) \times |f(w)| \tag{5}$$

Here, however, we can provide stronger guarantees, since: *i)* network costs are dominated by `quench` messages, and *ii)* these are disseminated by THRIFTY that has costs proportional to $|f(w)|$, as discussed later.

6.3 Parameters

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

HYBRID, PUREPOLL, and LAVISH. We simulate the variants HYBRID/40/20, HYBRID/30/14, and HYBRID/15/14. These parameter settings, as we later show, provide good coverage of tradeoffs in delay and cost, and we use them for comparison against the baselines. However, as explained in Sec. 4.1, 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 consider three corresponding variants: *i) "fast"*, PUREPOLL/ψ; *ii) "intermediate"*, PUREPOLL/$(\psi + \alpha/2)$; *iii) "slow"*, PUREPOLL/$(\psi + \alpha)$.

This in principle would yield nine PUREPOLL variants, $\delta \in \{15, 22, 29, 30, 37, 40, 44, 50, 60\}$. However, the variants $\{29, 37, 44\}$ are dropped as they are covered by similar values $\{30, 40\}$ yielding nearly identical performance. We also drop the variants with $\delta \geq 44$ because, as discussed later, they are too slow to be practical. Finally, to determine the point at which PUREPOLL overtakes HYBRID, we add the setting $\delta = 5$. Therefore, we consider the PUREPOLL variants with $\delta \in \{5, 15, 22, 30, 40\}$.

Finally, since LAVISH/ψ/α performs similar to HYBRID/ψ/α w.r.t. latency and monetary costs, and the other LAVISH variants perform closely for the values of ψ and α we consider, we simulate only LAVISH/15/14.

Bandwidth. Let $b(v) = b_{out}(v) + b_{in}(v)$ represent the *total* bandwidth made available by v for dissemination across all its friends, where $b_{out}(v)$ and $b_{in}(v)$ are the *total* inbound and outbound bandwidth. Our goal is to show that THRIFTY scales for nodes with up to 1 000 friends, so that we assign nodes with up to that many friends $b(v) = 100$ messages per second, split as $b_{out}(v) = 0.9 \times b(v) = 90$ and $b_{in}(v) = 0.1 \times b(v) = 10$. To ensure our simulations are meaningful, we constrain node v to allocate such bandwidth *evenly* amongst friends, i.e. $o_v(u) = b_{out}(v)/|f(v)|$ and $i_v(u) = b_{in}(v)/|f(v)|$ for all $u \in f(v)$. Since THRIFTY honors the bandwidth budget: *i)* the cost model we gave in Sec. 6.2 accurately represents bandwidth consumption; *ii)* the performance numbers we obtain are accurate.

For nodes with more than 1 000 friends we let bandwidth budgets grow "gracefully" at a rate of $100/1\,000 = 0.1$ per friend, i.e., we do not assume these nodes to be faster, per friend, than our worst-case scenario of 1 000 friends. To put numbers in context, a `quench` message takes 20 B for a UDP header, 64 bits for a timestamp plus identifier, and a variable-length bitmap for the message history. Bitmaps are sparse, since messages are confined to delay groups due to t_{out}, and those are small. With state-of-the-art compression techniques [14], ratios of 50% or more are a realistic expectation, leading to around 500 bits to represent an egonet with 1 000 members, and a total message size of 90 B. $b_{out}(v)$, in this case, would be 8.1 kB/s, and $b_{in}(v)$, 900 B/s.

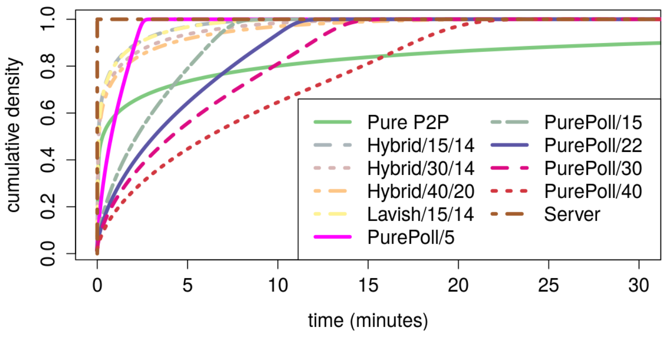


Fig. 7. Empirical CDF for the average receiver delay, $\mathbf{ard}$.

6.4 Results

Delays. Fig. 7 shows cumulative distribution functions (CDFs) for receiver delays of HYBRID and baselines, with complementary statistics in Table 1. The data confirms that PUREP2P suffers from significant performance issues, with the $\mathbf{ard}$ distribution having a long tail, reaching values as extreme as 49 days, and remaining nevertheless above 4.6h at the 99th percentile. We repeat the observation of [23] that this small 1% can translate into an unacceptable experience for millions of users in a system of the scale of Facebook. Instead, our cloud-based alternatives (HYBRID, PUREPOLL, LAVISH) effectively *solve the problem of the long delay tail* by putting a bound on $\mathbf{rd}$, as seen from the much smaller maximum and 99th percentile values w.r.t. PUREP2P.

Table 1 shows that HYBRID variants have maximum and 99th percentile $\mathbf{ard}$ values 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 420% and 285% faster, on average, than PUREPOLL/30 and PUREPOLL/15, respectively. While HYBRID outperforms its associated PUREPOLL variants, including the fast one, it provides a better experience for a significant fraction of the users across all parameter settings, even as we compare HYBRID/15/14 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 Fig. 7 as the nearly vertical shape of the CDF up until the 60th percentile—with the ability of mitigating the long delay tails of PUREPOLL.

Monetary cost. Yearly costs are shown in Fig. 8, based on Amazon S3 pricing [2]. Table 2 provides additional statistics.

The costs for HYBRID are generally lower than their associated PUREPOLL variants. Although these are attractive

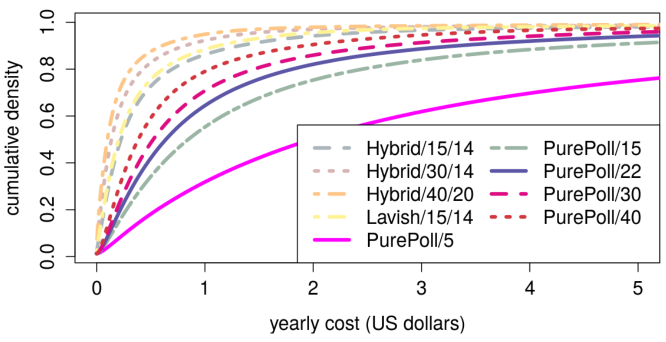


Fig. 8. Empirical CDF for yearly costs.

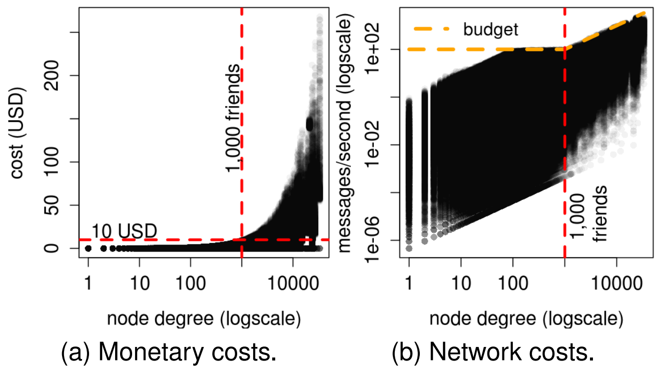


Fig. 9. Detailed cost plots for HYBRID/15/14.

(even at higher percentiles), maximum values can get rather high. A closer look at costs is provided in Fig. 9a, which shows a scatterplot of the estimated yearly cost versus node degree for HYBRID/15/14 (the other variants show a similar pattern). We can see that high values originate mostly from highly connected nodes, the most connected having 33 313 friends. For nodes with less than 1 000 friends, however, costs are no larger than 10$ a year for HYBRID/15/14, and even lower for HYBRID/30/14 and HYBRID/40/20 at 6$ and 4$, respectively. Finally, even if some users do get a large number of friends, users with huge ego networks are likely to trim friends they do not want to keep in touch so often, reducing costs considerably in practice.

As expected, LAVISH/15/14 incurs less costs than HYBRID/15/14 due to its faster (and unrealistic) `quench` dissemination. Cost improvements are again modest, however, and in the order of 3.3$/year at the 99% percentile, with statistically insignificant differences at the maximum value.

Network cost. Our last metric concerns the use of network resources. We analyze the average number of message/s processed at each node, using the cost models described earlier. We focus on `quench` messages since our main interest is on the base cost of the protocol. Indeed, the actual cost incurred by QUICK depends on the user's posting frequency and habits—variables out of our control and for which no model that correlates the required graph-structural properties and posting habits, without which numbers would be meaningless, exist. Yet, we argue that evaluating only `quench` overhead is still reasonable, since: 1) the bandwidth available for updates is ultimately given by what is available, minus the overhead we measure here; 2) user posts are so infrequent (see Sec. 2) when compared to `quench` messages that the averages we measure here are not affected by them. An analysis of the overhead of QUICK in the absence of posting frequencies can be found in [22].

All HYBRID variants present similar statistics for network costs (Table 3), supporting our claim (Sec. 5.3) that the dissemination layer is *saturated* in all cases. It also shows the importance of THRIFTY: network costs for LAVISH quickly spiral out of control, making it impractical. Costs for HYBRID are modest for most nodes (below 64 message/s) though maximum values are still high. Looking at the bandwidth/degree scatterplot for HYBRID/15/14 provided in Fig. 9b, however, we see that THRIFTY is doing its job: nodes with less than 1 000 friends are not going above budget, and nodes with more friends are nevertheless not allocating more than 0.1 message/s to the ego networks in which they participate. The high costs, therefore, are because some nodes simply have too many friends, and go beyond the scalability point we target. Finally, we note that such nodes are likely to be a problem to *any* OSN: even Facebook currently caps users to a maximum of 5 000 friends.

TABLE 1
Statistics for the average receiver delay **ard**
(s = second, m = minute, h = hour, d = day)

	avg.	50th	90th	99th	max
PUREP2P	18m	18s	31.7m	4.6h	49.2d
HYBRID/15/14	42s	4s	2.3m	7.3m	13.1m
HYBRID/30/14	1m	5s	3.3m	11.4m	20.8m
HYBRID/40/20	1.3m	6s	4.2m	14.8m	28.1m
LAVISH/15/14	43s	4s	2.3m	7.4m	13.1m
PUREPOLL/5	56s	46s	2.1m	2.6m	4.3m
PUREPOLL/15	2.7m	2.2m	6.3m	7.9m	14.3m
PUREPOLL/22	4m	3.1m	9.3m	11.3m	16.3m
PUREPOLL/30	5.2m	4.1m	11.9m	14.3m	17.7m
PUREPOLL/40	7.7m	6.2m	17.5m	21.3m	26m

TABLE 2
Yearly costs in USD.

	avg.	50th	90th	99th	max
HYBRID/15/14	0.84	0.23	1.32	11.29	268.02
HYBRID/30/14	0.51	0.14	0.81	7.01	171.83
HYBRID/40/20	0.37	0.10	0.60	5.21	128.82
LAVISH/15/14	0.69	0.20	1.12	8.97	265.14
PUREPOLL/5	7.39	1.98	11.64	100.01	1035.92
PUREPOLL/15	2.91	0.85	4.56	39.30	378.78
PUREPOLL/22	2.13	0.64	3.34	29.16	271.83
PUREPOLL/30	1.71	0.52	2.67	23.57	213.47
PUREPOLL/40	1.23	0.38	1.92	17.31	146.64

7 RELATED WORK

Decentralized OSNs have been around for almost a decade and carry an extensive body of published work [16], [28]. Hereafter, we focus on those approaches that best help the reader understand where our work stands in the literature.

Social Overlays. A distinguishing characteristic of our solution is its reliance on social overlays (SOs). SOs have been noted for their organic privacy and security properties which can either solve or otherwise mitigate fundamental trust-related issues in P2P systems [21], [32]. The fact that links and identities are hard to forge, and that nodes will not spontaneously connect to unknown participants means harmful exposure is drastically reduced, and that participants are more likely to cooperate. Indeed, the most widely used censorship-resistant content sharing network today, Freenet [41], relies on a SO for precisely these reasons.

A few other works in the decentralized OSN literature also adopt SOs. Safebook [6] leverages their trust properties along with source address rewriting to provide user pseudonimity when exposing profiles to strangers. Though not directly comparable to Clops, Safebook requires full 1-hop replication but does not specify a protocol to keep replicas in sync; Clops could provide this missing piece.

DiDuSoNet [13] employs a push gossip protocol for update dissemination over ego networks. Their push protocol differs from QUICK in that it biases selection towards central nodes and does not employ message histories. Though this may lead to local performance improvements, the fact that partitioning delays *dominate dissemination times* [23] means that DiDuSoNet will run into performance issues under churn. Our work addresses these issues, and could present a delay mitigation strategy to DiDuSoNet as well.

Profile Availability. One of the main concerns in P2P OSNs is reliably keeping profiles online while dealing with unreliable resources, both in terms of availability and trust. Works such as Lilliput [29], SOUP [15], and de Salve et al. [7] propose *mirror selection strategies* which help participants select and maintain a small (< 10) set of replicas so as to ensure availability. After selection, replicas must be kept in sync through a *replica synchronization* protocol.

Our work overlaps with these works in several ways. It can be seen as complementary on the one hand, as none of these works directly addresses actual dissemination of updates to friends. Indeed, whereas replica synchronization usually involves a handful of nodes, dissemination may involve hundreds of recipients, with problems that step firmly into the territory of P2P pub/sub. We could, therefore, conjugate our approach with existing work by replacing our use of S3 by, say, SOUP (or, equivalently, by complementing SOUP with Clops for update dissemination).

If we look at our work as a replication strategy, on the other hand, it trivially renders profiles always available to 1-hop friends, 100% of the time, even if we disregard copies residing in S3. We go one step further, however, and note that having replicas around is not enough if these are allowed to run stale; i.e., if u claims to have three replicas online, but two of those are storing snapshots from one year ago, then this really means u has only one useful replica online. Surprisingly, this is not something that gets much attention in related work—none of the solutions described here provides objective measurements of replica synchronization time, not even simple ones. Our work, therefore, could also be seen as an improved, bounded-latency replica synchronization protocol, even though it is not its main intent; i.e., if our goal were just to guarantee availability, we could do so with fewer replicas.

Pub/sub and Update Dissemination. Timely dissemination of updates can be seen as an instance of topic-based publish/subscribe (pub/sub) where each profile defines a topic to which both the profile owner and her friends are subscribed. In particular, disseminating only through subscribers requires a topic-connected overlay (TCO) [5], i.e., an overlay in which all users subscribed to a given topic form a connected subgraph. TCOs are useful to avoid spamming nodes with messages of no interest to them; social overlays are by construction TCOs for P2P OSNs.

TABLE 3
Average bandwidth usage, in message/s.

	avg.	50^{th}	90^{th}	99^{th}	**max**
HYBRID/15/14	8.19	2.95	15.75	63.51	2314.13
HYBRID/30/14	8.19	2.96	15.79	63.34	2298.39
HYBRID/40/20	8.19	2.95	15.80	63.23	2327.18
LAVISH/15/14	329.61	33.65	365.87	3828.24	45707320

The literature abounds with both general-purpose (e.g. [5], [31]) and OSN-specific solutions [27] to TCO pub/sub. All these solutions rely on the ability to rewire the overlay to optimize dissemination and/or robustness, and on subscription clustering (i.e., group nodes with similar subscriptions) to achieve approximate or deterministic topic connectivity while limiting node degrees. Clops, instead, must address pub/sub over an organic TCO—the social overlay—that *cannot* be rewired. Hence, we must derive all efficiency gains not from the overlay but from our protocols, which must adapt to heterogeneous clustering, widely differing node degrees, and transient partitions. This is a key difference of our work w.r.t. related ones.

Subscription clustering is also behind the efficiency gains in Cachet's pull protocol [26], which favors nodes with whom a node shares the most friends when fetching recent updates. Cachet's protocol can be seen as a pull counterpart to QUICK, though the former's reliance on DHTs means similarities with our work end there. Other works in the decentralized OSN literature, particularly early ones [4], [6], [12], either did not concern themselves with update dissemination, or suggested the adoption of simple, direct-mailing-like [8] protocols. These protocols, however, are known to be slow, and to face issues under churn [22].

Cloud-assisted OSNs. The idea of hybridizing a cloud and a serverless layer to reduce costs while ensuring QoS is not new. Confidant [20] mixes a P2P layer for storage with *active* aliases on the cloud that coordinate replicas and can be deployed under services such as Google's AppEngine. Vis-à-Vis [37] replaces departed or failed peers with stand-by nodes running in the cloud. Both approaches rely on active aliases, whereas we rely only on passive storage services. This represents a different type of exposure to the cloud provider (it is easier to encrypt data than memory contents) and tends to be costlier (storage costs less than servers). Mantle [11] proposes that user profiles be placed in cloud-based storage, as we do. Update dissemination is achieved by directly polling of the cloud service, however, putting Mantle in the same monetary cost/performance category as our baseline solution, PUREPOLL.

8 Conclusions

Social overlays, whose network links mirror friendship relationships, are an interesting option for building decentralized OSNs; however, their irregular and inflexible structure presents challenges to the implementation of key functionality such as fast dissemination of profile updates.

In this paper, we tackled these inefficiencies with Clops, a hybrid architecture that leverages the highly-available cloud infrastructure to adaptively support the social overlay, without sacrificing its key property of allowing communication only among friends. Clops relies on a specially-crafted dissemination layer that *i)* takes the social overlay structure into account, allowing quick update dissemination while respecting clustering and degree heterogeneity; *ii)* resorts to information stored in the cloud only when and where required. Our results show that Clops dramatically improves dissemination delays w.r.t. a pure serverless solution, and that monetary costs and network utilization remain acceptable for users with less than one thousand friends.

References

[1] C. Alexopoulos and A. F. Seila. Implementing the Batch Means Method in Simulation Experiments. In *Proc. of WSC'96*, 1996.

[2] Amazon.com, Inc. Amazon WebServices. https://aws.amazon.com. Visited 2017/05.

[3] F. Benevenuto et al. Characterizing User Behavior in Online Social Networks. In *Proc. of IMC'09*, 2009.

[4] S. Buchegger et al. PeerSoN: P2P social networking: early experiences and insights. In *Proc. of SNS'09*, 2009.

[5] C. Chen, H.-A. Jacobsen, and R. Vitenberg. Algorithms based on divide and conquer for topic-based publish/subscribe overlay design. *IEEE/ACM Trans. on Networking*, 2016.

[6] L. A. Cutillo, R. Molva, and M. Önen. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine*, 47(12), 2009.

[7] A. de Salve, P. Mori, L. Ricci, R. Al-Aaridhi, and K. Graffi. Privacy-Preserving Data Allocation in Decentralized Online Social Networks. In *Proc. of DAIS'16*, 2016.

[8] A. Demers et al. Epidemic algorithms for replicated database maintenance. In *Proc. of PODC'87*, 1987.

[9] Diaspora. https://joindiaspora.com/. Visited 2017/05.

[10] J. Douceur. The Sybil Attack. In *Proc. of IPTPS'01*, 2001.

[11] A. Famulari and A. Hecker. Mantle: A novel DOSN leveraging free storage and local software. In *Proc. of the 5th Intl. Conf. Adv. Infocomm Technology*, 2013.

[12] K. Graffi et al. LifeSocial.KOM: A secure and P2P-based solution for online social networks. In *Proc. of CCNC'11*, 2011.

[13] B. Guidi. *A distributed Dunbar-based Framework for Online Social Network*. PhD thesis, University of Pisa, 2015.

[14] T. Johnson. Performance measurements of compressed bitmap indices. In *Proc. of VLDB*, 1999.

[15] D. Koll, J. Li, and X. Fu. SOUP: An online social network by the people, for the people. In *Proc. of Middleware'14*, 2014.

[16] D. Koll, J. Li, and X. Fu. The Good Left Undone: Advances and Challenges in Decentralizing Online Social Networks. *Computer Communications*, 108, 2017.

[17] A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E*, 80(5), 2009.

[18] C.-Y. Lee. Correlations among centrality measures in complex networks. `arXiv:physics/060522`, 2006.

[19] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[20] D. Liu, A. Shakimov, R. Cáceres, A. Varshavsky, and L. P. Cox. Confidant: Protecting OSN Data without Locking it Up. In *Proc. of Middleware'11*, 2011.

[21] P. Maymounkonv. Tonika: social routing with organic security. http://pdos.csail.mit.edu/~petar/5ttt.org/, 2013. Visited 2017/05.

[22] G. Mega, A. Montresor, and G. P. Picco. Efficient Dissemination in Decentralized Social Networks. In *Proc. of P2P'11*, 2011.

[23] G. Mega, A. Montresor, and G. P. Picco. On Churn and Communication Delays in Social Overlays. In *Proc. of P2P'12*, 2012.

[24] G. Mega, A. Montresor, and G. P. Picco. Cloud-assisted Dissemination in Social Overlays. In *Proc. of P2P'13*, 2013.

[25] G. Mega, A. Montresor, and G. P. Picco. Social Overlays Meet the Cloud: A Hybrid Architecture for Profile Dissemination in Decentralized Social Networks: Supplemental material. 2017.

[26] S. Nilizadeh, S. Jahid, P. Mittal, N. Borisov, and A. Kapadia. Cachet: A Decentralized Architecture for Privacy Preserving Social Networking with Caching. In *Proc. of CoNEXT'12*, 2012.

[27] A. Olteanu and G. Pierre. Towards Robust and Scalable Peer-to-Peer Social Networks. In *Proc. of SNS'12*, 2012.

[28] T. Paul, A. Famulari, and T. Strufe. A survey on decentralized online social networks. *ACM Comput. Netw.*, 75, 2014.

[29] T. Paul, N. Lochschmidt, H. Salah, A. Datta, and T. Strufe. Lilliput: A storage service for lightweight peer-to-peer online social networks. In *Proc. of ICCCN'17*, 2017.

[30] T. Paul, D. Puscher, and T. Strufe. The user behavior in facebook and its development from 2009 until 2014. *CoRR*, abs/1505.04943, 2015.

[31] F. Rahimian, S. Girdzijauskas, A. Hossein Payberah, and S. Haridi. Vitis: A Gossip-based Hybrid Overlay for internet-scale Publish/Subscribe Enabling Rendezvous Routing in Unstructured Overlay Networks. In *Proc. Intl. Parallel & Distributed Processing Symposium (IPDPS'11)*, 2011.

[32] J. M. Rogers. *Private and Censorship-Resistant Communication over Public Networks*. PhD thesis, University College London, 2011.

[33] R. Roverso, S. El-Ansary, and S. Haridi. NATCracker: NAT combinations matter. In *Proc. of ICCN'09)*. IEEE, 2009.

[34] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011.

[35] S. Saroiu et al. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. of MMCN'02*, 2002.

[36] A. Shakimov et al. Privacy, Cost and Availability Tradeoffs in Decentralized OSNs. In *Proc. of WOSN'09*, 2009.

[37] A. Shakimov et al. Vis-á-Vis: Privacy-preserving online social networking via Virtual Individual Servers. In *Proc. of COMSNETS'11*, 2011.

[38] A. Singh, M. Castro, P. Druschel, and A. Rowstron. Defending against eclipse attacks on overlay networks. In *Proc. 11th ACM SIGOPS European Workshop (EW 11)*, 2004.

[39] A. Singh et al. Robust Overlays for Privacy-preserving Data Dissemination Over a Social Graph. In *Proc. of ICDCS'12*, 2012.

[40] M. Steiner, T. En-Jajjary, and E. W. Biersack. A Global View of KAD. In *Proc. of IMC'07*, 2007.

[41] The Freenet Project. https://freenetproject.org/. Visited 2017/05.

[42] Y. H. Wang. On the number of successes in independent trials. *Statistica Sinica*, 1993.

[43] Wikipedia contributors. Facebook-Cambridge Analytica Data Scandal — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Facebook%E2%80%93Cambridge_Analytica_data_scandal, 2018. Visited 2018/04.

[44] Z. Yao, D. Leonard, X. Wang, and D. Longuinov. Modeling Heterogeneous User Churn and Local Resilience of Unstructured P2P Networks. In *Proc. of ICNP'06*, 2006.

Giuliano Mega is a distributed systems specialist at SpazioDati s.r.l., an Italian startup specializing in the application of big data and semantics to business intelligence. Before that, he conducted his Ph.D. at the University of Trento (2008-2013) on understanding how to apply social overlays to decentralized online social networks.

Alberto Montresor is Associate Professor at the University of Trento since 2005. Before that, he has been with the University of Bologna (2002-2005). He has authored more than 80 papers on large-scale distributed systems, cloud computing and P2P networks. He is Associate Editor of Springer Computing and he served as Steering Committee Chair of the IEEE Conference on P2P Computing and as General Chair and Program Chair for DOA, DAIS, SASO, P2P.

Gian Pietro Picco is a Professor at the University of Trento, Italy. His research spans the fields of software engineering, middleware, and networking, and is oriented in particular towards wireless sensor networks, Internet of Things and Cyber-Physical systems, and large-scale distributed systems. He is currently an associate editor for ACM Trans. on Sensor Networks (TOSN), and co-Editor-in-Chief of the ACM Trans. on the Internet of Things (TIOT).