This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE P2P 2011 proceedings.

Cloudy Weather for P2P, with a Chance of Gossip

Alberto Montresor
University of Trento, Italy
alberto.montresor@unitn.it

Luca Abeni
University of Trento, Italy
luca.abeni@unitn.it

***Abstract*—Peer-to-peer (P2P) and cloud computing, two of the Internet trends of the last decade, hold similar promises: the (virtually) infinite availability of computing and storage resources. But there are important differences: the cloud provides highly-available resources, but at a cost; P2P resources are for free, but their availability is shaky. Several academic and commercial projects have explored the possibility of mixing the two, creating a large number of *peer-assisted* applications, particularly in the field of content distribution, where the cloud provides a highly-available and persistent service, while P2P resources are exploited for free whenever possible to reduce the economic cost. While executing active servers on elastic computing facilities like Amazon EC2 and pairing them with user-provided peers is definitely one way to go, this paper proposes a novel approach that further reduces the economic cost. Here, a passive storage service like Amazon S3 is exploited not only to distribute content to clients, but also to build and manage the P2P network linking them. An effort is made to guarantee that the read/write load imposed on the storage remains constant, regardless of the number of peers/clients. These two choices allows us to keep the monetary cost of the cloud always under control, in the presence of just one peer or with a million of them. We show the feasibility of our approach by discussing two cases studies for content distribution: the Dilbert's comic strips and the hourly News Update podcast from CNN.**

I. Introduction

Cloud computing represents an important enabling technology to compete in the world-wide ICT market. By providing the illusion of infinite computing resources, and more importantly, by eliminating upfront commitment, small- and medium-sized enterprises can play the same game as web behemoths like Microsoft, Google and Amazon.

The rise of cloud computing has progressively dimmed the interest in another Internet trend of the first decade of this century: the peer-to-peer (P2P) paradigm. P2P held similar promises with respect to cloud computing, but with important differences: while you cannot beat P2P from an economic point of view, the superior availability of the cloud (in the order of 99.99%) makes it a more believable environment for those who want to create novel web businesses and cannot afford to lose clients due to the best-effort philosophy of P2P [1].

Several academic and commercial projects have tried to mix the two paradigms to get the advantages of both: high-availability and low cost. The idea is to guarantee the former through the cloud, while aiming at the latter by exploiting, whenever possible, the inexpensive resources of peers. Two alternative philosophies have emerged:

- Adding peers to an existing centralized solution: to reduce the economic cost of storage, computing resources and more importantly bandwidth, the burden of providing the service is off-loaded to client peers: but, if and only if this does not affect service availability. Examples include video-on-demand systems [2] and more generally the field of file distribution [3].
- Augmenting P2P systems with cloud angels [4], i.e. elastic computing nodes with the specific task of satisfying requirements beyond the reach of a P2P network. Examples include bulk-synchronous content distribution services [4] and online P2P backup services [5].

This paper assumes content distribution as target application and proposes a third approach to *peer-assisted* distributed computing, with the purpose of further reducing the monetary cost. The approach is novel because all functionalities of our content distribution application are supported by a *passive* storage service such as Amazon S3: not only the actual information dissemination, but also the entire management of the P2P network (including node bootstrap, membership management and topology maintenance). No *active* elastic computing instances are required.

Having eliminated the fixed costs due to rented servers, the only remaining monetary costs are for storage and bandwidth. Storage consumption cannot be reduced, if durability is among the requirements. The focus should thus go on bandwidth: ideally, the number of accesses to the storage cloud should remain constant, regardless of the size of the P2P network.

Our proposed architecture, called CLOUDCAST, is based on two gossip protocols: one for topology bootstrap and membership management, and one for information dissemination over this topology. By carefully designing the former, we achieve our desired goal at both levels: the number of cloud accesses is one per gossip cycle, independently of the number of clients. The resulting system has the ability of scale both up and down: we rely on the availability of the cloud only when the P2P network is too small to provide a reliable service, shifting the load to the P2P network when additional resources are available. The monetary costs becomes negligible, in the order of few dollars per year. Adding more users does not cost a cent more; instead, the amortized cost per user decreases. Having few users is not a problem either: they will mostly exploit the cloud, but being few, their aggregated cost will be small.

In order to show the feasibility of this idea, we discuss two case studies based on content distribution: the delivery of the Dilbert's comic strips to 1.5M users, and the CNN News Update podcast to 40K users.

This work is supported by the Italian MIUR Project *Autonomous Security*, sponsored by the PRIN 2008 Programme.

978-1-4577-0149-8/11/$26.00

II. System Model

We consider a network consisting of a dynamic collection of *peers* that communicate through message exchanges. Each peer is uniquely identified by an ID (e.g. composed by IP address and port), required to communicate with that peer. The network is highly dynamic; new peers may join at any time, and existing peers may voluntarily leave or crash. Byzantine behavior is not considered.

Communication may incur unpredictable delays and is subject to failures. Single messages may be lost, links between pairs of peers may break; but we assume that the integrity of messages is not at risk.

An additional entity is the *storage cloud* C, which can be seen as a *(key,value)* store. A storage cloud is a passive element that can be accessed through $\langle \text{GET}, key \rangle$ and $\langle \text{PUT}, key, value \rangle$ operations, but cannot autonomously initiate communication [1].

Storing and retrieving data on the cloud is associated with a monetary cost. We assume the pricing model of Amazon S3 [6] on June 2011 as a reference example:

- GET/PUT operations cost 1μ\$ and 10μ\$ (microdollars);
- transfer-out/transfer-in data (measured in GB) cost 0.15\$ and 0.10\$, respectively;
- storage (measured in GB/month) costs 0.14\$.

We assume that the cloud provides adequate security mechanism to avoid unauthorized read/write operation. For example, in Amazon S3 each object can be associated with an access control list, and logging mechanisms are present to identify who actually requested an object. Furthermore, different payment models are available: apart from *owner-pays*, where the costs fall on the owner of a specific object, the *requester-pays* model allows to shift the costs toward the actual users requesting a piece of information.

III. Problem Statement

In this context, consider the problem of a single[1] source that wants to diffuse news updates to interested peers. Traditional solutions are either *cloud-based* or *P2P-based*.

A P2P-based solution allows peers to access other peers to retrieve the summary and the missing updates, thus using an "epidemic" diffusion mechanism that does not require centralized services. In order to use this approach:

1) Peers must know other peers that are currently online and could potentially help to diffuse updates; this is the *peer membership* problem. Two subproblems have to be considered:
   - a bootstrap mechanism is needed to allow peers to join the system, receiving the IDs of other peers and advertising its presence to them;
   - peers need to be organized in an overlay network to enable communication between them.
2) Peers must be able to pass the updates around, guaranteeing the eventual delivery of all updates to all participating peers; this is the *information diffusion* problem. Two separate aspects must be considered:
   - whenever a new update is available, peers currently online should receive it as soon as possible;
   - peers joining the system after an off-line period must be able to discover updates that have been generated during their absence.

The membership problem in P2P systems could be solved through a well-known host, which stores the IDs of all participating peers and helps in creating the appropriate overlay network, which can later be used for information diffusion. However, this approach would require a continuously available machine that could cost a relatively large amount of money, even if rented from a computing cloud (a "small instance" VM on Amazon EC2 costs 745$/year as of June 2011 [7]).

Cloud-based solutions are based on storing each update in the cloud, together with a summary item describing, in a compact way, the list of available updates (e.g., updates could be sequentially numbered and the summary could be the total counter). Interested peers may periodically read the summary and retrieve missing updates. Again, this solution can have a relevant monetary cost due to the high number of accesses to the cloud.

In this paper, we propose a hybrid approach that mixes P2P techniques with the usage of a storage (passive) cloud to address the problems highlighted above. The proposed approach takes the best of the two worlds to scale up, as well as scale down: this means that the news feed can be provided even in the presence of one single peer (which has no other option than using the cloud), as well as with millions of peers (in which case cloud operations are limited as much as possible, to keep the monetary cost under control). In other words, the algorithm presented in this paper is able to *adapt* the amount of cloud accesses depending on the number of peers present in the system (and on their stability/availability/reliability).

IV. Background

To understand how the dynamic adaptation mentioned above can be achieved, and how traditional P2P algorithms have to be modified to work with a passive cloud, a description of the algorithms and protocols used for P2P-based epidemic diffusion is needed. Hence, the original protocols are shortly recalled, starting with a quick introduction to the well-known epidemic protocols by Demers [8]. We dedicate a little more space to peer sampling, because it is less known and its modifications are at the heart of the CLOUDCAST approach. Additional details may found in the original papers [9], [8].

A. Information dissemination

The first epidemic protocols for information dissemination have been introduced more than 20 years ago [8], inspired by the spreading of epidemics and gossip rumors. Two styles of epidemic protocols have been proposed: rumor mongering and anti-entropy.

[1] We assume a single source since Amazon S3 only provides eventual consistency and multiple sources could incur in inconsistencies; nevertheless, this issue may be circumvented by the versioning mechanism of S3, whose details are beyond the scope of this paper.

In *rumor mongering*, peers are initially *ignorant*; when an update is *learned* by a peer, it becomes a *hot rumor*. While a peer holds a hot rumor, it periodically (every δ_{RUMOR} time units) chooses a random peer from the current population and pushes (sends) the rumor to it. When a peer p has tried to push a rumor m too many times, m stops being hot and it is retained by p without further pushing, to prevent unbounded dissemination. Among the variants defined by Demers et al. [8], we selected *coin* and *blind*, meaning that a rumor stops being hot with a probability p_{RUMOR} after each push operation, independently from the fact that the peers to which the update is sent is ignorant or not. This variant is the easier to implement and it is sufficient to demonstrate the viability of our approach.

In *anti-entropy*, each peer p periodically (every δ_{ENTROPY} time units) contacts a random partner q selected from the current population; p and q engage in an information exchange protocol where updates known to p but not to q are transferred from p to q (*push*), and/or viceversa (*pull*). Among the variants defined by Demers et al. [8], we selected *push-pull*, meaning that the information flows in both direction.

B. Peer sampling

Both the rumor mongering and anti-entropy descriptions make an explicit reference to the random selection of a gossip partner. Maintaining and diffusing a complete membership list to all participating peers from which to select may be prohibitive, due to network size and dynamism. Instead, a *peer sampling* service provides each peer with continuously up-to-date random samples of the entire population of peers. Such samples fulfill two purposes: they can be used by the epidemic broadcast service to obtain random peers, and they maintain a random topology connecting all peers, robust enough to deal with high levels of churn and even catastrophic failures.

Two potential candidates for a cloud-enabled peer sampling service are CYCLON [9] and NEWSCAST [10]. We adopt CYCLON because it has proven to be easier to modify for the inclusion of a storage cloud; NEWSCAST, on the other hand, is not suitable because it requires each node to periodically initiate gossip exchanges to advertise its availability. Being the storage cloud a passive component, this has ruled out NEWSCAST.

In CYCLON, each peer maintains a *partial view* of the system, i.e. a collection of *descriptors* representing a subset of the entire population of peers. Each peer periodically selects a partner to perform an epidemic information exchange, during which the oldest descriptors are discarded (potentially belonging to failed nodes), existing descriptors are shuffled between the exchange participants, and new descriptors are created (to advertise that the peer involved in the exchange are still active). Figure 1 contains the pseudo-code of the algorithm run by each node p, which is modeled by means of two distinct threads executed at each peer: the active one takes the initiative to communicate, while the passive one accepts incoming exchange messages.

Parameters of the CYCLON protocol are the maximum size of the partial view (called c) and the size of the messages sent

```
Active thread
repeat true
    wait δ_CYCLON time units
(1) % If space permit, re-insert old entries that have
      been sent in the previous cycle
    while view.size() < c and request.size() > 0 do
        r ← request.get(random(request.size()))
        view.put(r, min(view.get(r), request.get(r)))
        request.remove(r)
(2) % Increase age of all nodes in the view
    foreach r ∈ view do
        view.put(r, view.get(r) + 1)
(3) % Select oldest node in the view
    q ← arg max_r view.get(r)
    view.remove(q)
(4) % Fill up request, add a new descriptor for itself
    request.clear()
    repeat min(g − 1, view.size()) times
        r ← view.get(random(view.size()))
        request.put(r, view.get(r))
        view.remove(r)
    request.put(p, 0)
    res ← request.size()
(5) % Send a request message to q
    send ⟨REQ, p, request⟩ to q

Passive thread
on receive ⟨REQ, q, m⟩
(6) % Fill up reply and send it
    reply.clear()
    repeat min(g − 1, view.size()) times
        r ← view.get(random(view.size()))
        if r ≠ q then
            reply.put(r, view.get(r))
        view.remove(r)
    send ⟨REP, p, reply⟩ to q
(7) % Insert descriptors of m in view
    while view.size() < c − res and m.size() > 0 do
        r ← m.get(random(m.size()))
        view.put(r, min(view.get(r), m.get(r)))

on receive ⟨REP, q, m⟩
(8) % Insert descriptors of m in view
    res ← 0
    while view.size() < c and m.size() > 0 do
        r ← m.get(random(m.size()))
        view.put(r, min(view.get(r), m.get(r)))
```

Fig. 1. The CYCLON protocol executed by peer p

around (called g); they are both measured as number of descriptors included in them. Descriptors are pairs composed of a peer IDs and a timestamp representing the descriptor *age*. To simplify the pseudo-code, variable $view$ and messages $request$ and $reply$ are represented by a map data structure, associating a collection of keys (given by peer IDs) to their ages. The names of the methods operating on this data structure are taken from `java.util.Map`, and so their semantics.

The active thread is repeated periodically every δ_{CYCLON} time units (the *cycle length*). We postpone the explanation of code block (1) because depends on the rest of the algorithm. Code block (2) increases the age of all descriptors by one; new descriptors will be created with age 0. In code block (3), the oldest descriptor in the view is selected (the one with largest age; in case of multiple descriptors with the same age, the selection is random among them) and removed from the view. The removal action is performed to "clean" the overlay from old descriptors which could belong to failed peers. Code block (4) prepares message $request$ by randomly extracting $g-1$ descriptors from the local view and finally adding a fresh descriptor (age 0) for p which acts as a "node p is still alive" announcement. After code blocks (3) and (4), the view contains now $c-g$ descriptors. Having an active request sent out, $request$.size() slots of the view are reserved until we receive a reply. This is indicated by assigning the value $request$.size() to variable res. Finally, p sends this message to q as an active request in code block (5).

The other thread passively waits for incoming messages. If m is a request from q, a reply message is created and sent to q in code block (6), by removing g random descriptors from the current view. If m is a reply, res is set to 0, to indicate that no active request is pending any more. In code block (7) and (8), all the descriptors received from q are finally inserted in $view$, paying attention to not insert two descriptors for the same node, but instead taking the freshest one (i.e., the one with smaller age). Note that if m is a request, no more than $c-res$ entries may be present in the view, to avoid to occupy reserved slots. If m is a reply, the reserved slots can be used.

At this point, we are in the position to explain code block (1). At the beginning of each cycle, it is possible that some of the c slots of $view$ are not occupied; this happens either when duplicate peer IDs are detected between the descriptors received and those already present in the view, or when a request message is not answered (either because the message is lost, or because the destination has crashed). To avoid to permanently shrink the view, descriptors which have been removed could be re-inserted if space permits.

Combining the two threads, the final effect is a continuous random shuffling of the views of peers participating in the protocol, with requests and replies exchanged among them.

The protocol provides high quality (i.e., sufficiently random) samples and maintains a random overlay topology which is robustly connected not only during normal operation (with relatively low churn), but also during massive churn and even after catastrophic failures (up to 80% peers may fail), quickly removing failed peers from the local views of correct peers [9].

V. Architecture

As previously stated, CLOUDCAST is based on a modification of two basic protocols: *peer sampling* [11] and *epidemic broadcast* [8], described in Section IV. A storage cloud is used to allow such protocols to work even when few peers are present in the system, to address the bootstrap problem without having a centralized server which is always active, and to tolerate extreme churn. Hence, the two protocols have been opportunely modified to cope with peculiarities of our scenario: the storage cloud participates as a member, but being a passive component (a key/value store), its active role is played by the peers interacting with it.

A. Basic CLOUDCAST architecture

As in traditional P2P systems, peer sampling is used to efficiently address the peer membership problem, while epidemic broadcast is used to diffuse information. Fig. 2 contains an architectural summary of CLOUDCAST. Normal peers and the cloud maintain the same information (updates to be diffused and partial views of the system membership), but peers execute active protocols (peer sampling and epidemic broadcast with the two subprotocols rumor-mongering and anti-entropy) that are not present in the passive storage cloud. Interactions among the components are shown with labeled arrows.

In CLOUDCAST, rumor mongering is used to push updates as fast as possible towards the peers currently online. A cycle length smaller than the one used for anti-entropy can be adopted, because rumor mongering requires fewer resources. On the other hand, there is a non-zero chance that an update will not reach all peers because it prematurely stops being hot. To guarantee the eventual delivery of updates, CLOUDCAST augments rumor mongering with a less frequent anti-entropy protocol. Anti-entropy also enables peers joining the network to receive updates created during their absence.

B. Storage clouds vs epidemic protocols

One of the most important requirements in CLOUDCAST is to reduce the number of accesses to the cloud as much as possible. This result can be achieved by ensuring that the total number of descriptors pointing to the cloud (the cloud in-degree) is not too high; on the other hand, the cloud in-degree cannot be too small, otherwise references to the cloud risk to be lost and the system works as a traditional P2P system. By using CYCLON as a peer sampling protocol, it is possible to ensure that the in-degree in_p of each peer p (total number of descriptors pointing to p) tends to stay around c (note that the out-degree $out_p = c$ is constant by design). The reason is as follows: at any given time, timestamps are uniformly distributed. The probability for a given descriptor of being selected by $removeOldest()$ is $1/c$. During a cycle, the expected number of descriptors pointing to p selected (and then removed) is thus in_p/c. During the same cycle, one new descriptor for p is created by p itself. If, during a cycle, $in_p > c$, the expected number of p's descriptors that are removed from the network is larger than 1, and in_p tends to decrease; if $in_p < c$ is smaller than c, in_p tends to increase.

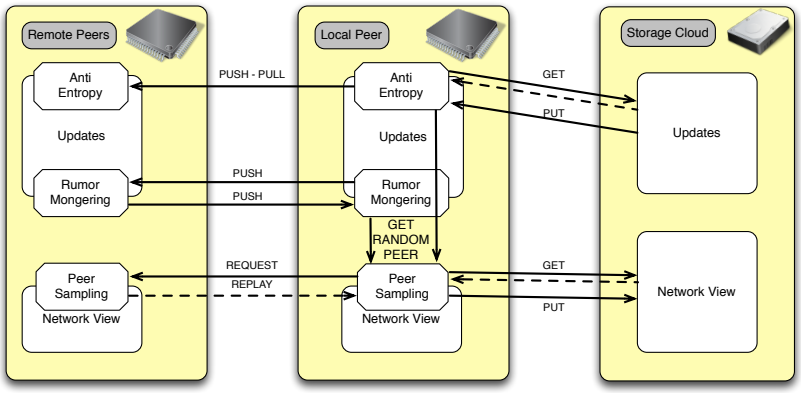

Fig. 2. The CLOUDCAST architecture.

Finally, if $in_p = c$, the in-degree tends to remain stable. Hence, if we add the cloud C as a peer to a sufficiently large network (with the size $N > c$), then the expected value of in_C is c.

Following the same reasoning as above, the expected number of contacts to the cloud originating from all peers is $c/c = 1$ per cycle (for both peer sampling and epidemic broadcast), independently of the size of the network. In other words, the load per unit of time imposed on the cloud is constant and CLOUDCAST scales up without problems. If the number of peers is $n < c$, the expected number of contacts is n/n, proving that CLOUDCAST scales down as well.

Note, however, that CYCLON cannot be used without modifications, for various reasons. The first reason is that since the cloud is a key/value store, it cannot perform autonomous computations; hence, a peer p interacting with the cloud must play the cloud role as well. A second reason is that the unmodified CYCLON protocol does not work well when only few peers are active in the system. Moreover, the number of cloud descriptors risk to be too small (smaller than c, and even going to zero). This problem can be due, for example, to the fact that references to the cloud progressively disappear (even if CLOUDCAST never deletes a cloud descriptor after a cloud contact) due to churn (a peer with a cloud descriptor crashes).

C. The CLOUDCAST protocol

Peer sampling. To cope with the fact that the cloud has no active behavior, each peer p must perform a GET operation on C to retrieve the *entire* view from the cloud; then p takes the responsibility of shuffling both its view and the cloud's one, to finally write the new cloud on the view through a PUT operation.

The modified cloud protocol is shown in Figure 3. The code of block (4-5) substitutes blocks (4) and (5) of Figure 1. If the oldest descriptor q is a reference to the cloud, a GET message is sent to C; the cloud will eventually reply with a VIEW message, as described in the second part of Figure 3.

Code blocks (9)–(12) emulate a gossip exchange between peer p and the passive storage cloud. p extracts $g-1$ random elements from the local view and puts them in the emulated *request* message and adds a fresh descriptor for itself. On the other hand, one or more references to the cloud may be added to the emulated *reply* message (see below when and why this should happen) and thus random descriptors are extracted from the view v received from the cloud and put in *reply* until the message size is g. At this point, the two emulated messages *reply* and *request* contain g descriptors each and are inserted in v and *view*, respectively, with the effect of shuffling the local view and the cloud view. At this point, the new cloud view is written in the storage cloud in code block (13). Given the random shuffling of the views, concurrent updates to the cloud do not cause any particular inconsistency. This has been confirmed experimentally.

The random shuffling performed by p when interacting with the cloud follows the CYCLON protocol, with one notable exception: given that new descriptors are created in the active thread (that is absent in the cloud), existing cloud descriptors would be progressively forgotten and in_C would go to 0. To solve this problem, peers that interact with the cloud substitute the cloud descriptor with a fresh one, mimicking the active thread of the cloud. We call this operation "refreshing".

Dealing with failures. To cope with potentially disappearing cloud references (for example, due to churn or message losses), we exploit the fact that GET operations are associated with metadata, including a last-modified field, which can be used to detect whether the expected number of contacts per cycle is respected. Approximate time synchronization between cloud clients and the cloud is already required. Being t the time of the last contact and k a threshold parameter, if $\mathsf{now}() - t > \delta_{\text{CYCLON}} \cdot k$ then p creates a new cloud descriptor in addition to refreshing the existing one (low contact frequency means to a low number of cloud descriptors); if $\mathsf{now}() - t < \delta_{\text{CYCLON}}/k$ the cloud descriptor is not refreshed, but simply removed (high contact frequency means high number of cloud descriptors).

Recovery mechanism. Unfortunately, exceptional conditions like high churn and message loss rates may still wipe all

Modifications to the active thread of CYCLON

```
(4–5) % Act differently weather q is the cloud ID or not
      if q = C then
          send ⟨GET, "view"⟩ to C
      else
          % Execute code block (4)
          % Execute code block (5)
```

Addition to the passive thread

```
      on receive ⟨VIEW, v, t⟩
(9)       % Fill up request
          request.clear()
          request.put(p, 0)
          while request.size() < g and view.size() > 0 do
              r ← view.get(random(view.size()))
              request.put(r, view.get(r))
              view.remove(r)
(10)      % Fill up reply
          reply.clear()
          if now() − t > k · δ_CYCLON then
              reply.put(C, 0)
          if now() − t > δ_CYCLON/k then
              reply.put(C, 0)
          while reply.size() < g and v.size() > 0 do
              r ← v.get(random(v.size()))
              if r ≠ p then
                  reply.put(r, v.get(r))
              v.remove(r)
(11)      % Insert descriptors of reply in view
          while view.size() < c and reply.size() > 0 do
              r ← reply.get(random(reply.size()))
              view.put(r, min(view.get(r), reply.get(r)))
              reply.remove(r)
(12)      % Insert descriptors of request in v
          while v.size() < c and request.size() > 0 do
              r ← request.get(random(request.size()))
              v.put(r, min(v.get(r), request.get(r)))
              request.remove(r)
(13)      % Write the new view in the cloud
          send ⟨PUT, "view", v⟩ to C
```

Fig. 3. The modified CYCLON protocol executed by peer p

the cloud references from the system. At that point, no one will ever contact the cloud again. As a recovery mechanism, each node p maintains a variable *last* containing the most recent round p heard about a cloud reference. This value is piggybacked on normal CYCLON messages; p updates *last* whenever it receives a cloud reference in its view, or it receives a value of *last* from another peer which is more up-to-date with respect to its current value. If all cloud references disappear, eventually *last* will stop increasing at all nodes. A node whose *last* variable has not changed for t rounds, creates a new cloud reference with probability p_{RECOVERY}.

Bootstrapping. A beneficial side effect of adopting the storage cloud is that we can also easily solve the bootstrap problem: when joining the system, peers perform a GET operation and retrieve a first view, that can be later used to start the normal CYCLON protocol. If the view retrieved from the cloud is smaller than c, this means that $n < c$, and a cloud descriptor is added.

Information diffusion. At the epidemic broadcast level, each new update is recorded in the cloud with a PUT operation; an update counter is also updated. The update becomes a hot rumor, and a rumor mongering protocol is started; this protocol never involves the cloud. In the anti-entropy protocol, on the other hand, the peer contacting the cloud reads the update counter and retrieves missing updates, if any. Given that in S3 keys are user-assigned, it is easy to devise a mechanism to translate update numbers into unique keys.

TABLE I
PARAMETERS USED IN THE EVALUATION.

Parameter	Value	Meaning
n	2^6–2^{16}	Total number of peers
δ_{cyclon}	10s	Cycle length of CYCLON
δ_{rumor}	1s	Cycle length of rumor mongering
$\delta_{entropy}$	10s	Cycle length of anti-entropy
c	20	View size
g	5	CYCLON message size
p_{RUMOR}	0.2	Probability of becoming removed
k	4	Threshold parameter
t	20	Silent period
p_{RECOVERY}	0.1	Recovery probability

VI. EVALUATION

CLOUDCAST has been evaluated through an extensive number of simulations based on the event-based version of PEERSIM [12]. We provide here only a few insights about the good behavior of the protocol. Further studies are needed to explore the parameter space at depth; for now, we set them to reasonable values summarized in Table I, partially taken from the original CYCLON paper [9]. If not differently stated, each experiment is repeated 50 times, over which aggregate values are computed. When graphically feasible, standard deviation is represented by vertical bars.

A. Evaluating peer sampling

The first question to ask is: is our modified CYCLON protocol capable to maintain an approximately constant in-degree of the cloud, in spite of the network variation in size? Fig. 4 shows what happens when the network size oscillates between 1 and 500 peers in a 1-day period. The cloud in-degree is shown with respect to time. When the network grows from 1 to 20 peers, the in-degree grows accordingly; this means that all peers have a cloud descriptor. As the size increases up to 500 peers and then decreases, the in-degree remains in the range $[5, 35]$, to finally fall down to zero when

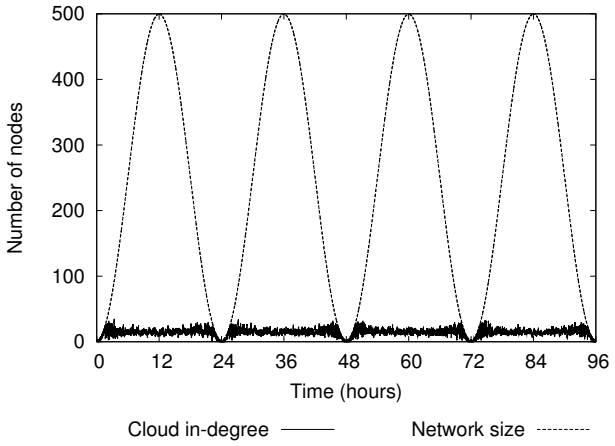

Fig. 4. Cloud in-degree in case of a network that oscillates daily between 0 and 500 peers. Four days (96 hours) are shown.

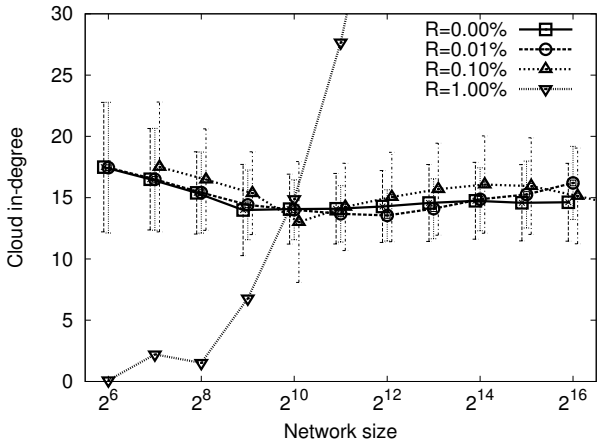

Fig. 6. Ability to maintain the correct in-degree under different levels of churn R, on a network with variable size between 2^6 and 2^{16}.

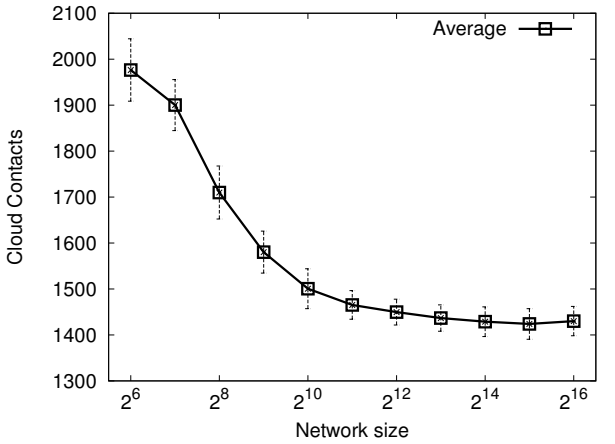

Fig. 5. Number of GET/PUT operations per day on the cloud generated by CYCLON, on a network with variable size between 2^6 and 2^{16}.

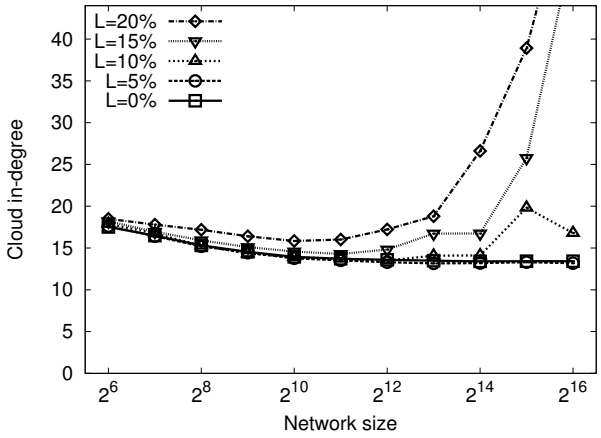

Fig. 7. Ability to maintain the correct in-degree under different levels of message loss L, on a network with variable size between 2^6 to 2^{16}.

all the peers leave. This behavior repeats periodically; a single experiment lasting four days is shown.

The choice of 500 peers is motivated by the graphical representation; but we are interested in much larger networks. Fig. 5 shows the total number of cloud GET/PUT operations performed by CYCLON over the period of one day, for increasing sizes of the network. These experimental results confirm that the cloud load can be maintained under control.

The capability of dealing with failures is illustrated by the next two figures. Fig. 6 shows the average in-degree under different churn conditions, in network whose size varies between 2^6 to 2^{16} nodes. Each experiment last for 6 simulated hours, for a total of 300 hours. A churn rate R of $p\%$ means that at each second, each peer has a $p\%$ probability of abruptly leaving the network and being substituted by a new peer.

The lines corresponding to 0.00%, 0.01% and 0.10% show approximately the same behavior, with an average in-degree around 15. When $R = 1.00\%$, however, the system "goes crazy" and the number of churn descriptors grows enormously. This behavior is caused by the fact that the mechanism that adds a new cloud descriptor whenever a new node joins the system creates too many new descriptors that cannot be eliminated in time by the cleaning mechanism of CYCLON. It must be noted, however, that a churn level of 1.00% corresponds to an expected peer life time of less than 2 minutes, which is two order of magnitudes larger than a reasonable level of churn in P2P systems [13].

Beside churn, another source of unreliability is the loss of messages. Given the sporadic interactions among peers, it is perfectly reasonable to assume that communication is based on UDP (rather than TCP) and is subject to failures. For this reason, we have tested the behavior of CYCLON under different levels of message loss L, from 0% to 20%. Results are shown in Fig. 7. As before, we assumed that the cloud is reliable; moreover, that communication with it is not subject to failures (a reasonable assumption, as the communication with the cloud is based on TCP). For low levels of message losses (10% or less), there is no measurable effect on the average cloud in-degree; the in-degree grows with the message loss rate, but not to unreasonable levels.

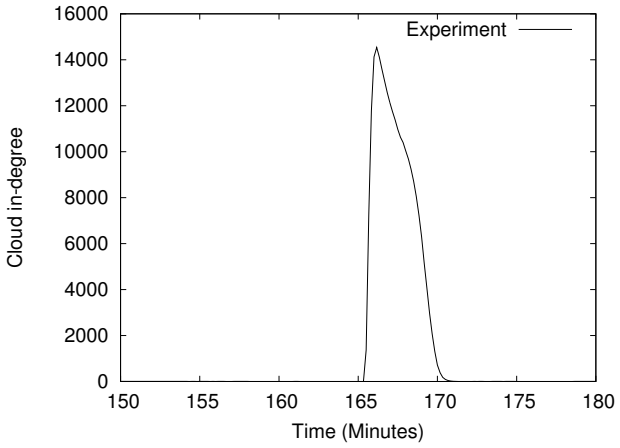

Fig. 8. The recovery mechanism in action. A single run is shown (message loss rate $L = 20\%$ and network size $N = 2^{16}$), where in_C goes to zero at minute 165 and the correct number is recovered by minute 171.

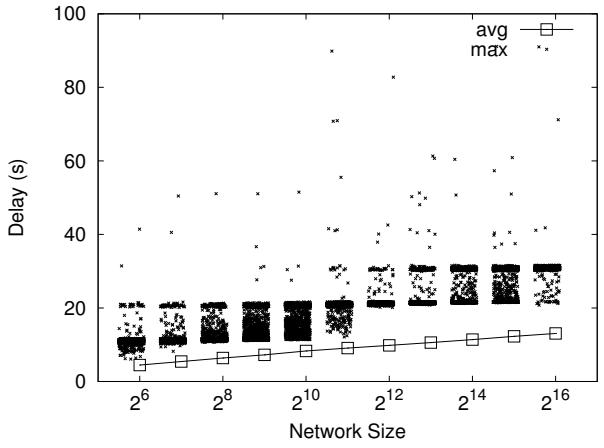

Fig. 9. Average and maximum delay of messages. A total of $24 \cdot 60 = 1440$ messages sent over the period of one simulated day, on a network with variable size between 2^6 and 2^{16}.

But average values over a period of 300 hours must be taken with care. Figure 8 shows an excerpt of a single experiment with message loss rate $L = 20\%$ and network size $N = 2^{16}$. At minute 165, the number of cloud references goes to zero; the recovery mechanism kicks in, and in few cycles the number of references becomes as large as $\frac{1}{4}$ the size of the network. By minute 171, however, the situation is back to normal. Note that such an exceptional behavior never happens with $L = 0\%$ and $L = 5\%$, and happened 2, 13 and 16 times with $L = 10\%$, $L = 15\%$ and $L = 20\%$, respectively (over a period of 300 simulated hours). While this behavior may hint for serious problems in the protocol, note that message loss rates larger than 10% are rather exceptional; furthermore,the number of cloud GET/PUT operations is still proportional to the average in-degree, which remains approximately constant as shown in Fig. 5.

B. Evaluating message diffusion

To evaluate CLOUDCAST, we simulated networks of variable sizes for the period of one simulated day, during which one message is sent every minute. In all our experiments, all the messages are eventually delivered by all correct peers. Fig. 9 shows the average delay of these messages. The average is represented by a straight line, and is computed by averaging the arrival time at each peer for all messages. The maximum delay represents the time needed to deliver a single message to all peers; here, each individual message is represented by a distinct point. The larger the size, the larger the portion of peers that receive the message through anti-entropy (cycle length 10s) rather than rumor mongering (cycle length 1s); this explain the black patterns around multiples of $10s$. We believe that both average and maximum represent adequate results.

Fig. 10 shows the behavior of CLOUDCAST over a dynamic network. 48 hours are shown. Every 4 hours, an update is generated for a total of 16 updates. The network size oscillates between 0 and 1000 nodes (shown on the right y-axis). The dashed lines represent the average delay of each of these

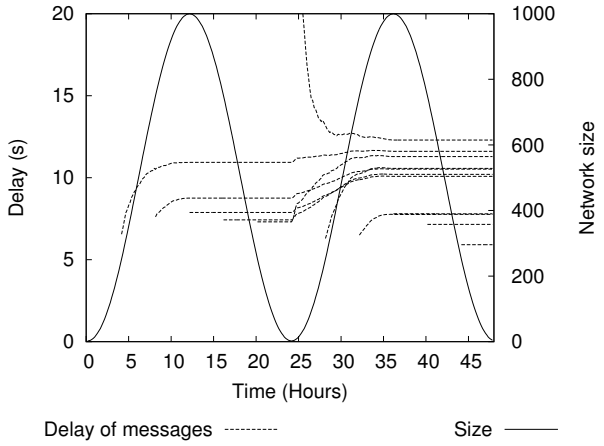

Fig. 10. Behavior of CLOUDCAST in a network whose size oscillates daily between 0 and 1000 nodes (size on the right y-axis). Updates are generated every 4 hours and their average delays is shown (delay on the left y-axis).

updates; the delay of update m at node p is computed as the interval between the generation of m and the receipt of m by p, if p was operational when m has been generated; or the time passing between p joining the system and the receipt of m by p, if p was offline when m has been generated. Different behaviors can be seen: updates generated when the network is growing are quickly distributed among nodes that are online through rumor-mongering, to be later distributed through anti-entropy to nodes that join the system. Updates generated when the network is shrinking are distributed just through rumor-mongering. Updates generated when the network has size 0 (e.g., at hour 24) and is growing again are initially distributed just through anti-entropy, so their delay is initially high, but quickly decreases.

Another aspect of CLOUDCAST to be evaluated is the overhead generated by P2P communication, measured as the average number of times a single update is exchanged among peers, shown in Fig. 11. This value is strongly dependent

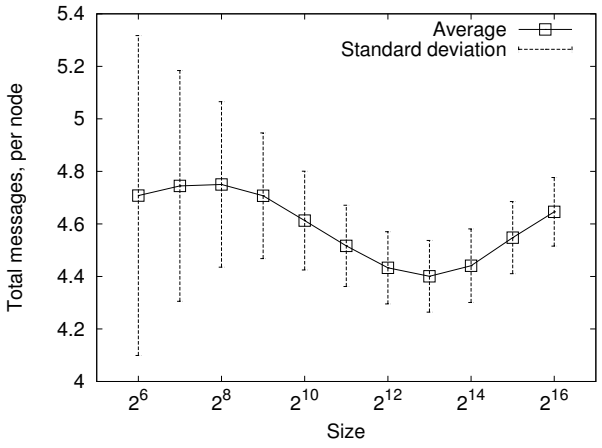

Fig. 11. Average number of updates sent/received per node. A total of $24 \cdot 60 = 1440$ updates generated over the period of one simulated day.

on the selected rumor mongering strategy (e.g., blind/coin versus feedback/counter), the values of configuration parameters like p_{RUMOR}, the ratio between δ_{RUMOR} and δ_{ENTROPY}, and the maximum delay that can be tolerated. As stated before, we selected reasonable values for all these parameters and we obtained acceptable maximum delays and communication overhead. Our goal is not to find the best trade-off between delay and overhead, but rather demonstrate the feasibility of our approach.

Some figures of merit are not reported in a graphical way, because they are better discussed analytically. For example, the communication overhead of CYCLON (i.e., the bandwidth consumed by peers to run the peer sampling service) is given by the size of view messages divided by the cycle length δ_{CYCLON}. We assume that each descriptor takes 10B (4 for the IP address, 2 for the port number, 4 for the timestamp). A view with 20 entries, plus a counter, plus TCP/IP headers requires 240B. The expected number of messages to be sent every 10s (the length δ_{CYCLON}) is 2 (one request plus one expected reply), bringing to an overhead of 48B/s.

C. Case studies

It is time now to give concrete numbers, and understand the monetary cost of our protocol. We consider two cases studies: Dilbert's comic strips and the hourly News Update podcasts from CNN, delivered to their respective user base. Dilbert's is a popular comic strip read online by more than 1.5 millions unique visitors.[2]. One new strip (a GIF of 50KB in size) is issued per day. No statistics are available for the number of listeners of this podcast; as a very conservative estimate, we use 1% of the number of followers of the CNN Twitter account[3] (1% of 4.0 millions followers = 40,000). Each update is 1MB.

Serving the content of these feeds through Amazon S3, without using the CLOUDCAST approach incurs in the following costs:

- Dilbert: $50\text{KB} \cdot 365 \cdot 1.5\text{M} = 27.375\text{TB}$; at the cost of 0.15$ per GB, this corresponds to $4,106$$.
- CNN: $1\text{MB} \cdot 365 \cdot 24 \cdot 40\text{K} = 350.400\text{TB}$; at the cost of 0.15$ per GB, this corresponds to $52,560$$.

These costs are just for payload bandwidth and thus optimistic: given that we do not want to specify how information is accessed (how many strips/podcasts per request?), we did not compute the overhead for each request.

Now, consider the costs associated to the peer sampling service and to the information diffusion service. While a view with 20 entries requires 200B, we approximate it to 1KB to include REST API overhead, additional view data, etc. Up to $2,000$ contacts per day (Fig. 5) times 365 days means $730,000$ GET/PUT operations per year. Given the cost model of Section II, these corresponds to 0.73$ and 7.3$, respectively. Transfer-in and transfer-out are equal to $1\text{KB} \cdot 730,000 = 0.73\text{GB}$, corresponding to 0.11$ and 0.07$ respectively. The total cost of CYCLON is thus 8.21$ per year. This amount does not include the cost of peers joining the system; this depends on the usage pattern. We just recall that 1$ pays 10^6 GET operations $\equiv 10^6$ joins.

In CLOUDCAST, the cost depends on the number and the size of news. The number of cloud operations is equal to CYCLON, but obtaining the update counter only requires GET operations, with a cost of 0.73$. A peer p downloads an update from the cloud only if the update counter implies that p has not received the update through rumor mongering. Since Fig. 9 shows a maximum delay around 100s, in this period a maximum of 10 anti-entropy cycles can be executed (given that $\delta_{\text{ENTROPY}} = 10$s). The expected number of anti-entropy requests in 10 cycles is 10, meaning that each update will be downloaded 10 times from the cloud in the worst case. In our case studies:

- Dilbert: The total number of strips read from the cloud is $365 \cdot 10 = 3,650$. The transfer-out is thus $3,650 \cdot 50\text{KB} = 0.18\text{GB}$, which corresponds to 0.03$.
- CNN: The total number of podcasts read from the cloud is $365 \cdot 24 \cdot 10 = 87,600$. Considering 1MB per tweet, this corresponds to a total transfer-out of 87,6GB, which corresponds to 13.14$.

The total yearly costs for our two case studies are less than 25$, a tiny fraction of the costs required to serve the entire content through the cloud. Please note that the overhead assumed in this figures is an extremely pessimistic overestimate of the real value.

VII. Related work

The usage of passive peers in P2P systems is not new: for example, the Gnutella Web Caching System [14] solves the bootstrap problem with a set of web servers that store the IP addresses of active peers, so that joining peers can easily find other peers to connect with. Other systems, such as p2pvpn [15], use a BitTorrent tracker similarly to a host

[2] www.thefreelibrary.com/Dilbert+at+20-a0197405344
[3] www.twittercounter.com

cache. Other attempts at solving the bootstrap problem without using a centralized server are based on network scanning or probing [16], [17], [18].

The difference between CLOUDCAST and these systems is that the storage cloud is directly integrated in the gossip protocol, at all levels: in the bootstrap phase, to maintain the overlay connected in the presence of strong churn, to provide the content of messages when only few peers are available, etc. Moreover, CLOUDCAST keeps the number of accesses (and hence the cost) under control. In this sense, CLOUDCAST is more suitable for small P2P systems without a large budget. Other systems proposed a P2P approach for implementing a distributed storage service, or use storage clouds to implement content delivery networks [19], but to the authors' best knowledge CLOUDCAST is the first protocol that enhances storage clouds with epidemic protocols.

The modifications that CLOUDCAST applies to the traditional gossip paradigm are particularly important, because when introducing passive and highly-available elements like a storage cloud in a P2P system, such system is not homogeneous anymore, and this can affect the performance of the gossip protocols [20].

VIII. Conclusions and Future Work

By making a concrete example, this paper shows a novel distributed programming paradigm that mix the dependability of cloud computing with the low cost of P2P networks. Even in its simplicity, the example shows the enormous potentiality behind this approach.

CLOUDCAST has been tested in different scenarios, showing that the proposed solution is effective in maintaining the overlay connected, while quickly diffusing messages (even to peer that dynamically join the system).

This paper is meant to demonstrate the viability of this approach. We discuss here future work that could make this application more realistic and further reduce the monetary cost of our approach:

- Distinct epidemic broadcast strategies should be evaluated. The current approach (rumor mongering blind/coin, anti-entropy) is the simplest to implement; alternative strategies could easily outperform the current one.
- In the case of a large network, the number of operations performed on the cloud by the anti-entropy protocol could be easily reduced, by waiting to receive an update from other peers and postponing the access to the cloud itself. The size of the network could be easily computed in a decentralized way [21], [22].
- It is interesting to note that a largest bill in our protocol is due to peer sampling, rather than the actual download of updates. As above, strategies that adapt to the size of the network could reduce enormously the number of costly PUT operations on the cloud.

A real implementation of CLOUDCAST for GNU/Linux has been developed, based on the GRAPES library [23]. Such an implementation is under evaluation on realistic settings in the Internet. Meanwhile, the source code and the configuration files required to reproduce these results are available at `http://peersim.sf.net/code/Cloudcast.zip`.

References

[1] "Amazon S3 web page," http://aws.amazon.com/s3/, Amazon Web Services LLC.

[2] C. Huang, J. Li, and K. Ross, "Peer-assisted VoD: Making internet video distribution cheap," in *Proc. of the 6th Int. Workshop on Peer-to-Peer Systems (IPTPS'07)*, Bellevue, WA, USA, Feb. 2007.

[3] B. Cohen, "Incentives build robustness in bittorrent," in *Proc. of the 1st Workshop on Economics of P2P Systems*, 2003.

[4] R. Sweha, V. Ishakian, and A. Bestavros, "Angels in the cloud: A peer-assisted bulk-synchronous content distribution service," Computer Science Department, Boston University, Tech. Rep. BUCS-TR-2010-024, 2010.

[5] L. Toka, M. Dell'Amico, and P. Michiardi, "Online data backup: a peer-assisted approach," in *Proc. of the IEEE Int. Conference on Peer-to-Peer Computing (P2P'10)*, Delft, The Netherlands, Aug. 2010.

[6] "Amazon S3 pricing," http://aws.amazon.com/s3/pricing, Amazon Web Services LLC.

[7] "Amazon EC2 pricing," http://aws.amazon.com/ec2/pricing, Amazon Web Services LLC.

[8] A. Demers *et al.*, "Epidemic algorithms for replicated database maintenance," in *Proc. of the 6th ACM Symp. on Principles of Distributed Computing (PODC'87)*. Vancouver, BC, Canada: ACM Press, Aug. 1987, pp. 1–12.

[9] S. Voulgaris, D. Gavidia, and M. Van Steen, "CYCLON: Inexpensive membership management for unstructured P2P overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

[10] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "The peer sampling service: Experimental evaluation of unstructured gossip-based implementations," in *Proc. of Middleware 2004*, ser. LNCS, vol. 3231. Springer, 2004.

[11] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Trans. Comput. Syst.*, vol. 25, no. 3, p. 8, Aug. 2007.

[12] A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," in *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, Seattle, WA, Sep. 2009, pp. 99–100.

[13] M. Castro, M. Costa, and A. Rowstron, "Performance and dependability of structured peer-to-peer overlays," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*. IEEE Computer Society, 2004.

[14] "Gnutella web caching system," http://www.gnucleus.com/gwebcache/specs.html.

[15] "P2PVPN," http://www.p2pvpn.org.

[16] C. Gauthier Dickey and C. Grothoff, "Bootstrapping of peer-to-peer networks," in *Proc. of the Int. Symposium on Applications and the Internet (SAINT'08)*, 2008, pp. 205–208.

[17] C. Cramer, K. Kutzner, and T. Fuhrmann, "Bootstrapping locality-aware P2P networks," in *Proc. of the 12th IEEE Int. Conf. on Networks (ICON'04)*, 2004.

[18] J. Dinger and O. Waldhorst, "Decentralized bootstrapping of P2P systems: A practical view," in *Proc. of Networking 2009*. Springer, 2009, pp. 703–715.

[19] J. Broberg and Z. Tari, "MetaCDN: Harnessing storage clouds for high performance content delivery," in *Proc. of The Sixth Int. Conf. on Service-Oriented Computing (ICSOC'08)*, 2008.

[20] N. Tölgyesi and M. Jelasity, "Adaptive peer sampling with Newscast," in *Proc. of the 15th Int. Euro-Par Conf. on Parallel Processing (Euro-Par'09)*. Delft, The Netherlands: Springer-Verlag, 2009, pp. 523–534.

[21] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 1, pp. 219–252, Aug. 2005.

[22] A. Montresor and A. Ghodsi, "Towards robust peer counting," in *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, Seattle, WA, Sep. 2009, pp. 143–146.

[23] L. Abeni, C. Kiraly, A. Russo, M. Biazzini, and R. L. Cigno, "Design and implementation of a generic library for p2p streaming," in *Proc. of the Workshop on Advanced Video Streaming Techniques for Peer-to-Peer Networks and Social Networking*, Florence, Italy, October 2010.