2010-08-12

# Cloud-based Content Distribution on a Budget

Albanese, Francesco

CS Department, Boston University

# Cloud-based Content Distribution on a Budget

*Boston University, Computer Science Technical Report BUCS-TR-2010-022*


Francesco Albanese
Eurecom
Sophia Antipolis, France
albanese@eurecom.fr

Damiano Carra
University of Verona
Verona, Italy
damiano.carra@univr.it

Pietro Michiardi
Eurecom
Sophia Antipolis, France
michiard@eurecom.fr

Azer Bestavros
Boston University
Boston, USA
best@bu.edu



***Abstract*—To leverage the elastic nature of cloud computing, a solution provider must be able to accurately gauge demand for its offering. For applications that involve *swarm-to-cloud* interactions, gauging such demand is not straightforward. In this paper, we propose a general framework, analyze a mathematical model, and present a prototype implementation of a canonical swarm-to-cloud application, namely peer-assisted content delivery. Our system – called CYCLOPS – dynamically adjusts the off-cloud bandwidth consumed by content servers (which represents the bulk of the provider's cost) to feed a set of swarming clients, based on a feedback signal that gauges the real-time health of the swarm. Our extensive evaluation of CYCLOPS in a variety of settings – including controlled PlanetLab and live Internet experiments involving thousands of users – show significant reduction in content distribution costs (by as much as two orders of magnitude) when compared to non-feedback-based swarming solutions, with minor impact on content delivery times.**


## I. INTRODUCTION

*Cloud computing* has emerged as a compelling paradigm for deploying Information and Communication Technology (ICT) solutions on the Internet, because it enables solution providers to easily scale up or down, or migrate their offerings seamlessly across resources – compute servers, storage, platforms, and services – offered by one or more cloud providers, yielding significant cost savings due to economies of scale. More importantly, the elasticity of the "pay-as-you-go" paradigm enables solution providers to reign in operating costs, especially when demand is highly dynamic, or unpredictable. For many cloud-based ICT solutions, gauging demand is straightforward. For instance, a cloud-based web hosting/caching solution can easily gauge demand – and hence scale up or down its use of elastic cloud resources – by observing the number (or average response time) of its web transactions.

Increasingly, however, cloud-based solutions are evolving from simple *client-to-cloud* interactions (reminiscent of the traditional client-server model) into *swarm-to-cloud* interactions, wherein the cloud-based solution is not merely responding to individual client requests, but rather to the collective demand of a "swarm" of clients, making the determination of what constitutes demand for cloud resources for purposes of elastic resource allocation far more complicated. In this paper, we propose a general framework and present a prototype implementation that enable elasticity for a canonical "swarm-to-cloud" application – namely peer-assisted content delivery.

**Towards Elastic Cloud-Based, Peer-Assisted CDNs:** Traditional Content Delivery Networks (CDNs) such as Akamai [3] were conceived as special-purpose clouds catering almost exclusively to large, highly-popular content providers such as iTunes and CNN. Today, the advent of cloud-based storage and delivery solutions such as Amazon S3 [1] and CloudFront [1] make it possible for much smaller-scale content providers to deploy and elastically provision their own cloud-based CDNs in an almost real-time fashion. The major cost contributor for such cloud-based CDNs is *off-cloud bandwidth*: the bandwidth consumed to deliver content from the CDN content servers (in the cloud) to the CDN clients (off the cloud). To reduce off-cloud bandwidth, an increasing number of CDN solutions (including those offered by major market players such as Akamai [3], Limelight [5], and Amazon [1]) rely on swarm-based, peer-assisted approaches that leverage the uplink capacity of end-users to reduce off-cloud bandwidth consumption. This approach, which is particularly effective for highly-popular content, can be seen as seamlessly bridging client-to-cloud and swarm-to-cloud interactions: For less-popular content, a cloud-based, peer-assisted CDN behaves as a traditional (client-server) CDN system, whereas for highly-popular content, it behaves as a peer-to-peer system.

Existing cloud-based peer-assisted CDNs rely on swarm-based protocols such as BitTorrent [4]. While such protocols are quite efficient for exchanging content among peers (in terms of download time, resource utilization, and fairness), they are not designed to provide the content source with the means to gauge the marginal utility of its contribution to the swarm. Specifically, in our cloud-based peer-assisted CDN setting, swarm-based protocols do not enable the content server (in the cloud) to gauge and hence manage the inherent tradeoffs between off-cloud bandwidth utilization and the efficacy of content delivery. This is precisely the capability that the work presented in this paper aims to provide.

**Paper Scope and Contributions:** We present a novel framework for cloud-based peer-assisted CDN solutions in which the content server (inside the cloud) is able to adjust the off-cloud bandwidth it contributes to the swarm (the set of clients outside the cloud) so as to achieve a specific *objective* based on a *feedback signal* related to the state of the swarm. Our framework is general enough to allow for many possible combinations of objectives and feedback signals. For instance, the objective may simply be to keep the swarm alive based on a feedback signal indicating the level of redundancy for particular pieces of content in the swarm. Alternately, the

objective may be to ensure a desirable level of service based on a feedback signal gauging average delivery time to clients.

To establish a reference model for these as well as other combinations of objectives and feedback signals, in Section II, we develop an analytical model that quantifies the cost-performance tradeoff for cloud-based, peer-assisted content delivery. Our model relates off-cloud bandwidth utilization (the cost incurred by the provider) to the average delivery time (the performance observed by clients). Along these lines, our findings suggest the existence of a quiescent (close to optimal) operating point beyond which the marginal utility from additional off-cloud bandwidth utilization is negligible.

Armed with this understanding, in Section III, we present the design and prototype implementation of CYCLOPS, a peer-assisted content delivery cloud service. The content server in CYCLOPS is able to modulate its bandwidth contribution to the swarm so as to remain in the vicinity of the aforementioned quiescent operating point – thus minimizing its cost without sacrificing performance. Our design relies on the feedback signal provided through an on-line monitoring tool, which we have implemented as part of CYCLOPS.

To demonstrate the effectiveness of our approach, in Sections IV and V we report on a fairly extensive series of Internet experiments, in which we compare the performance of CYCLOPS to those of "open-loop" swarm-based protocols used by cloud-based content delivery services. Our experiments are carried out both in a controlled environment (by delivering content to PlanetLab clients) and in the wild (by delivering content to a real Internet user population). These experiments show that our feedback-based approach reduces drastically the volume of data served from the cloud (and hence the cost incurred by the content provider) with negligible performance degradation. More to the point, in live experiments involving more than 10,000 users exhibiting highly dynamic arrival and departure patterns, we were able to document monetary savings of up to two orders of magnitudes for our system.

## II. MODELING THE COST-PERFORMANCE TRADEOFF

In this Section we develop a model that relates off-cloud bandwidth utilization by a content server in the cloud to the average delivery time perceived by a set of swarming users (clients) outside the cloud.

We consider a dynamic environment, where clients join a swarm, download the content, and eventually leave the system. The number of clients in the swarm is not known *a priori*, but it can be characterized by arrival and departure rates. While these rates may fluctuate drastically,[1] we assume that for the content download timescale (say minutes) they remain constant, allowing the system to reach a steady state in which the arrival and departure rates equalize, and consequently the average number of clients in the swarm is constant.

Let $N$ be the steady-state average number of clients in the swarm, and let the content be divided into $M$ independent pieces. If $M \gg 1$ then a client holds $M/2$ pieces on average. For analytical tractability, we do not model network bottlenecks or losses.

Consider a *birth-death Markov chain* whose state $s_k$ represents $k$, the number of replicas of a single (arbitrary) piece of content.[2] For a generic state $s_k$, there are two possible transitions: (1) either the piece is replicated, resulting in a *piece birth*, and thus a transition from state $s_k$ to state $s_{k+1}$, or (2) a client holding a replica of the piece leaves the swarm and is replaced by a new client that does not have the piece, resulting in a *piece death*, and thus a transition from state $s_k$ to state $s_{k-1}$.

Let $\alpha_k$ indicate the *average* rate at which the content server injects a piece in the swarm at state $s_k$. Let $\lambda$ denote the piece replenishment rate resulting from client contributions: $\lambda$ is computed by dividing the aggregate upload capacity of all $N$ clients by the total number of pieces $M$. Both $\alpha_k$ and $\lambda$ are expressed in pieces per second.

We assume a random piece replication strategy.[3] Thus, the probability of choosing to replicate the particular piece (modeled by the Markov chain) out of the $M/2$ pieces available at the client, is $2/M$. The probability that no client will choose to replicate that piece is $(1-2/M)^k$, since $k$ is the number of clients holding the piece in state $s_k$. This yields a probability of $1-(1-2/M)^k$ for going from state $s_k$ to state $s_{k+1}$.

To compute the transition rate from state $s_k$ to state $s_{k+1}$ we must also account for the rate $\alpha_k$ at which the content server *independently* injects the piece into the swarm. This yields a transition rate of $\lambda \cdot (1-(1-2/M)^k) + \alpha_k$. Notice that state $s_0$ is a special state in which only the content server can inject the piece. Thus, the transition rate from state $s_0$ to state $s_1$ is equal to the server upload rate $\alpha_0$.

Let $\mu$ denote the client departure rate (measured in clients per second). The probability of a death out of state $s_k$ is the probability that any one of the $k$ clients holding the piece leaves the swarm. The probability that a given departure is by one of these $k$ users is $k/N$. Thus, the transition rate from state $s_k$ to state $s_{k-1}$ is given by $\mu k/N$.

In summary, the transition rates from state $s_k$ to state $s_{k'}$, denoted by $s_{k,k'}$, can be expressed as follows:

$$s_{k,k'} = \begin{cases} \alpha_0 & \text{if } k=0 \text{ and } k'=1 \\ \lambda\cdot(1-(1-2/M)^k)+\alpha_k & \text{if } k'=k+1,\ 0<k<N \\ \mu k/N & \text{if } k'=k-1,\ 0<k\leq N \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

We now compute the probability $\pi_0$ to be in state $s_0$. For simplicity, we consider the case in which the content server uploads a piece at an average rate $\alpha_k = \alpha$, $\forall k$, irrespectively

[1] Such fluctuations are typical for "hot" viral Internet content, which gets published, gains significant popularity fairly quickly, but eventually dies off over time.

[2] One can envision an identical, independently evolving Markov chain for each one of the $M$ pieces that make up the content.

[3] In contrast to more sophisticated replication strategies [7], random piece selection simplifies analysis and provides conservative performance bounds.

of its state; by solving the Markov chain we get:

$$\pi_0 = \left[1 + \frac{\alpha}{\mu} N\,(1 + \Phi)\right]^{-1} \tag{2}$$

where

$$\Phi = \sum_{k=2}^{N} \left(\frac{N}{\mu}\right)^{k-1} \frac{1}{k!} \prod_{i=1}^{k-1} \left[\lambda\left(1 - \left(1 - \frac{2}{M}\right)^{i}\right) + \alpha\right]$$

We now proceed to finding the relationship between the average server rate $\alpha$ and the mean download time. Each client obtains $1/N$ of the swarm's upload capacity, which is $M(\lambda + \alpha)$. Since the content is composed of $M$ pieces, the mean download time can be computed as $T = M/(M(\lambda+\alpha)/N) = N/(\lambda+\alpha)$. This is true as long as the probability of being in state $s_0$ is small enough. If this probability increases, then we have an additional term for the mean time spent in state $s_0$: this can be computed by multiplying the probability of state $s_0$ ($\pi_0$) by the time spent in state $s_0$ ($1/\alpha$). Hence, the mean download time is bounded by:

$$T \leq \frac{N}{\lambda + \alpha} + \frac{\pi_0}{\alpha}, \tag{3}$$

To illustrate the utility of this model, consider a swarm of $N = 100$ clients downloading content consisting of $M = 2000$ pieces, with a mean client upload rate of $\lambda = 10$ pieces per second, and a client departure rate of $\mu = 0.5$ clients per second. Figure 1 shows the average download time as a function of the server upload rate, as predicated by Equation 3.

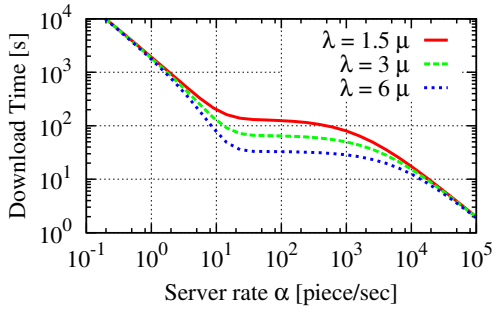



Fig. 1. Mean download time as a function of the server rate ($N = 100$, $M = 2000$, $\mu = 0.5$).

Figure 1 quantifies the tradeoff between the off-cloud bandwidth utilization (*i.e.*, the average upload rate $\alpha$ of the content server) and the average delivery rate to clients involved in a swarm with upload capacity $\lambda$. It shows three operating regions. The first operating region (left-side of the plot) is when $\alpha$ tends to zero, resulting in piece starvation, and a corresponding increase in download time. The second operating region (right-side of the plot) is when $\alpha$ tends to values that far exceed $\lambda$, resulting in a client-server-like mode of operation. The third and more interesting operating region is an intermediate one,[4] within which an increase in $\alpha$ does not result in a corresponding decrease in download time.

The behavior predicted by our model suggests the existence of a quiescent operating point (at the transition between the first and second operating regions depicted in Figure 1), beyond which the marginal utility from additional off-cloud bandwidth utilization is negligible. A content server operating around this quiescent point would be fully leveraging the uplink bandwidth of its clients, while minimizing its own cost: *operating below this quiescent point would jeopardize performance, and operating above this quiescent point would be cost inefficient.*

Armed with this observation, we are now ready to describe the design and prototype implementation of a content server that uses a feedback signal to adjust its bandwidth contribution to the swarm so as to remain in the vicinity of a nominal quiescent operating point. While our framework allows for many combinations of objectives and feedback signals, in the remainder of this paper we focus on the objective of maximizing the performance per unit cost, using the availability of content in the swarm as the feedback signal.

## III. System Design and Implementation

We now present the design of Cyclops, our cloud-based peer-assisted content delivery service.[5]

### A. Overview of Cyclops

As depicted in Figure 2, our Cyclops service consists of a *content server* and a *swarm monitor*, both residing in the cloud. The swarm monitor interprets the signaling messages exchanged between swarming clients, and generates a feedback signal that enables the content server to gauge the marginal utility of its contribution to the swarm. The content server participates in the swarming protocol to satisfy client requests, but only *feeds* the swarm when its contribution is deemed necessary (based on the feedback signal). In Cyclops, the swarm feeding rate is set to maximize the swarm performance-per-unit-cost, using the availability of content in the swarm as the feedback signal. As established in our model in Section II, the quiescent operating point for this objective is the minimum rate that avoids swarm starvation.

Cyclops is conceived to work with *any* swarm-based application/protocol that features (1) a coordinating entity that tracks all swarm participants, enabling them to establish peer-to-peer connections; (2) content that is divided into pieces to be distributed/exchanged independently; and (3) a control messaging scheme used by swarm participants to advertise piece availability.

For practical reasons, we present our system and conduct our experiments focusing on a single content server, used to deliver a single content (file) to a set of clients. Problems related to concurrent swarms are orthogonal to our approach, and the solutions proposed in the literature, *e.g.*, [17], can be integrated independently. Similarly, issues related to the

[4]The "width" of this region depends on the health of the swarm, which is a function of the content popularity captured by the client arrival/departure rate $\mu$, and the mean client upload bandwidth $\lambda$. For the particular settings used in Figure 1, this intermediate region is given by $\alpha \in [10, 1000]$ piece/sec.

[5]Our Cyclops service can be seen as injecting bursts of content into a swarm of clients, just as in Greek mythology the primordial one-eyed giant Cyclopes were the source of Zeus' thunderbolts.

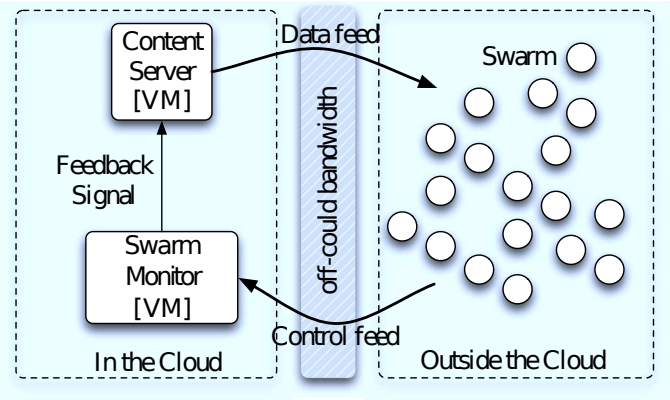



Fig. 2. Overview of CYCLOPS Architecture: The content server and swarm monitor reside in the cloud in distinct virtual machines, with off-cloud bandwidth used for data feed (to the swarm) and control feed (from the swarm).

efficiency of the distribution process, solved using approaches based on traffic locality, are complementary to our solution, and previous work on this topic, *e.g.*, [9], [10], can be incorporated seamlessly.

CYCLOPS was conceived and implemented as a cloud service that can be deployed on existing cloud platforms. Specifically, we focused on the Amazon Web Services (AWS) environment, and produced an Amazon Machine Image (AMI) that supports *both* the content server and the swarm monitor functionalities.[6]

### B. The CYCLOPS Swarm Monitor

Swarm monitoring in CYCLOPS is achieved using a set of components residing in the cloud, called the On-line Feedback (OF) nodes. OF nodes connect to a live swarm, but neither download nor upload content: they monitor *all* clients in the swarm and collect signaling messages they exchange. Using this information, OF nodes construct snapshots in time that characterize the health/performance of the swarm. In our particular implementation, these snapshots are used to derive the instantaneous piece availability, which constitutes the feedback signal fed to the CYCLOPS content server using a complementary protocol.

To ensure scalability (and seamless elasticity), we adopted a distributed design for OF nodes, whereby new clients joining the swarm are assigned to OF node to balance load. Accordingly, a swarm $S$ is partitioned into $N_p$ non-overlapping sets, where $N_p$ is the number of OF nodes in the system. Swarm partitioning is achieved using consistent hashing [13]: each OF node is responsible for a fraction of the key-space, defined by the client ID (*e.g.*, IP address).

### C. The CYCLOPS Content Server

The main objective of the content server is to minimize off-cloud bandwidth consumption without running the risk of starving the swarm. Based on the feedback signal provided by the swarm monitor, the content server feeds the swarm only when necessary, *i.e.*, when piece availability falls below a desirable threshold. To that end, in our design we adopted an ON/OFF control strategy, whereby the content server operation oscillates between two states: *serving* and *idle*.

When in the *serving state*, the content server dedicates its full uplink capacity to serve missing pieces of content. By design, the server avoids injecting duplicate pieces into the swarm. The rationale for doing so is that pieces can be quickly replicated by the swarm participants themselves. All clients connected to the content server are induced to request the set of missing pieces,[7] which constitute the *serving set* maintained by the content server. This serving set is partitioned into $k$ non-overlapping subsets that are announced as "available." For instance, if the serving set consists of pieces $\{1,2,3,4\}$ and $k = 2$, then $k$ messages each announcing pieces $\{1,2\}$ and $\{3,4\}$, respectively, will be sent to $k$ users that will eventually issue download requests. Once a piece has been served, it is removed from the serving set, provided that the swarm monitor has confirmed the presence of the piece in the swarm. When the server has finished injecting all missing pieces into the swarm, it transitions to the *idle state*.

When in the *idle state*, the content server simply closes all connections to remote clients, and refuses any incoming connection. The content server remains in the idle state until the feedback signal triggers a transition to the *serving state*.

## IV. EXPERIMENTAL METHODOLOGY AND SETUP

In this section, we summarize the specifics of the CYCLOPS instance we have experimented with, along with various details regarding deployment on a commercial cloud. We also describe the three types of experiments we have conducted: two were in a controlled environment (involving PlanetLab clients under our control), and the third was in the wild (involving thousands of real Internet users accessing content we advertised and made available).

**BitTorrent-based Swarming:** As we alluded to in Section III, CYCLOPS can be instantiated to work with any swarm-based content distribution protocol, supporting a specific set of features. For experimental purposes, we created an instance of CYCLOPS that is compatible with the popular BitTorrent (BT) client.[8] This choice is partly motivated by the wide adoption of BT by Internet users, as well as its adoption by many cloud-based content delivery services (including Amazon S3 and many others [2]) as an underlying swarming protocol. The details of the BT protocol and algorithms are not essential to understanding CYCLOPS, thus we refer interested readers to [16] for a technical description of BT. Here we only mention that the coordinating entity that maintains the list of clients in the swarm is called the tracker, and that the two control messages used by BT to advertise pieces available at a client are the "have" and the "bitfield" messages: they indicate the

[6]Upon publication of this work, we will release to the research community the CYCLOPS AMI, along with set-up and configuration instructions.

[7]This is possible since the server masquerades as a set of virtual clients holding a fraction of all available pieces.

[8]In all experiments, clients execute unmodified BT code.

availability at a client of a specific (single) piece, and of a set of pieces, respectively [16].

In the remainder of this paper, we use open-loop-BT to refer to an "open-loop" BitTorrent swarm-assisted content delivery system, whereas we use CYCLOPS to refer to our "feedback-controlled" BitTorrent swarm-assisted content delivery system.

**Deployment Details:** We used Amazon's Elastic Computing Cloud (EC2) to host, on separate virtual machines, the open-loop-BT content server (called the seed) and tracker, and the CYCLOPS content server and swarm monitor.[9] To mitigate the negative impacts on networking performance due to shared resources (CPU and I/O) in a virtualized environment, we used large EC2 instances, which were all located in a *single* US-based data center. Our open-loop-BT and CYCLOPS content servers were well-provisioned, with an upload capacity of 2.4 Mbps.

**Flash Crowd Experiments:** To emulate a flash crowd arrival process, we deployed a set of clients on PlanetLab machines, whereby all clients initiate their requests as a result of a centralized trigger: clients start downloading the content within 1 minute of that trigger signal. Once a user is done downloading the content it continues to serve other clients until the end of the experiment. We conducted our experiments using two flash crowd sizes of $L = 50$ and $L = 300$ clients, respectively. In order to minimize the resource utilization of PlanetLab nodes, we used a homogeneous configuration with an application level cap of 160 Kbps for the client's uplink capacity, which is the default setting for BT. The content size was set to 50 MB.

**Waves of Arrivals Experiments:** We synthesized extreme swarm dynamics on PlanetLab, with the goal of studying CYCLOPS under stress. The dynamics consisted of three successive bursts of client arrivals: a first burst of 100 clients arrive in a 10-minute span and leave after completing their download (within 50 minutes of arrival); a second burst of 100 clients join the swarm just before the mass exodus of the first wave of users. This process is then repeated for a third burst of arrivals. The interval between the mass exodus from one wave and the burst of arrivals from the next wave is set up in such a way that there would not be sufficient time for content pieces to propagate fully from the clients of one wave to the next (which should cause the swarm monitor's feedback signal to trigger the CYCLOPS content server to rev up its contribution to the swarm). As before, the client's uplink capacity was capped at 160 Kbps, and the content size was set to 50 MB.

**Live Internet Experiments:** We conducted experiments to evaluate our system under realistic CDN operating conditions, including web-driven arrival and departure processes for users drawn from a diverse set of ISPs and with diverse software settings. To do so, we prepared a 350MB file that we named after a popular TV-series. We created two distinct *torrent* meta-files (one for distribution using CYCLOPS and the other for distribution using open-loop-BT), and we publicized both simultaneously on popular content search web-sites, including *isohunt*, *mininova* and *btjunkie*. We took particular care in publicizing the two torrents exactly on the day of their TV broadcast. In these experiments, both the CYCLOPS and the open-loop-BT content servers had no cap on their uplink capacity (beyond what is possible through a large EC2 instance), and needless to say, we had no control on the settings (or even the BT variants) of the clients.

**Performance Metrics:** In all of our experiments, we considered two main performance metrics. From the content server perspective, we measured the aggregate volume of data uploaded during an experiment, *i.e.*, the off-cloud bandwidth utilization. Since content servers are under our control, we can measure their bandwidth utilization using local log files. From the client side, we measured the content delivery times. For PlanetLab experiments, we did that by collecting application-level logs from the clients. For live experiments, where we do not have access to client logs, we measured the content delivery times using our swarm monitor, which aggregates information provided by OF nodes. The accuracy of this approach was validated using the PlanetLab experiments.[10] To assert the statistical significance of our results, our PlanetLab experiments were performed five times for each configuration.

## V. EXPERIMENTAL RESULTS

### A. Flash Crowd Experiments

End-users' performance in downloading content is expressed in terms of individual download times. Figure 3 reports the most important percentiles (25th, 50th and 75th) of the empirical cumulative distribution function (ECDF) of download times.

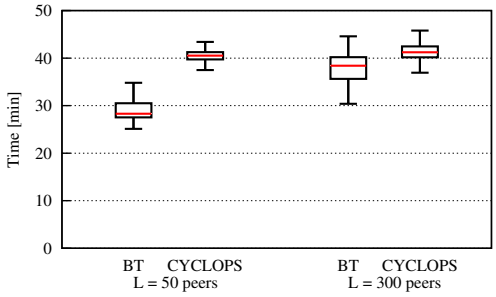



Fig. 3. **Flash Crowd**: content download times (file size: 50MB).

As a general trend, we observe that the median download time of open-loop-BT swarms is lower than that of CYCLOPS swarms, with the gap reduced in larger swarms.[11] The reason lies in the fact that an open-loop-BT seed keeps feeding the swarm during the whole experiment, resulting in a larger

[9]In our experiments, a single OF node proved to be sufficient to monitor the entire swarm fed by CYCLOPS.

[10]We compared the download times computed using individual log files (of PlanetLab clients) to those obtained from OF nodes, and verified the match between the empirical cumulative distribution functions of download times for the two methodologies.

[11]Aside from visible but relatively small variations, the download time for CYCLOPS clients was less sensitive to the swarm size.

fraction of users receiving data from the content server itself (which is faster than the user), and hence the shorter content delivery time.

TABLE I
**FLASH CROWD**: AVERAGE SERVER LOAD (FILE SIZE: 50MB)

| | BT | CYCLOPS |
|---|---|---|
| $L = 50$ | 12.2 | 1 |
| $L = 300$ | 15.36 | 1 |

The above explanation is further confirmed by the results in Table I, which reports the average off-cloud resource utilization expressed in volume of data served by both the CYCLOPS and the open-loop-BT content servers, normalized by content size. An open-loop-BT seed injects the swarm with 10–15 times the size of the original content, whereas CYCLOPS feeds the swarm only when necessary, which given the static nature of this experiment is once. These results corroborate the intuition discussed in Section II. A content server that can gauge the marginal utility of its contribution to a swarm can settle in the vicinity of an operating point in which an additional expense of off-cloud resources has a marginal effect on the swarm performance.

### *B. Waves of Arrivals Experiments*

Figure 4 shows the key percentiles of the empirical cumulative distribution function (ECDF) for the delivery times experienced by clients in the successive waves of arrivals. In this case, the difference between the delivery times achieved by CYCLOPS and the open-loop-BT content servers is small: the median value of the distribution indicates an advantage of roughly 15% in favor of the latter.

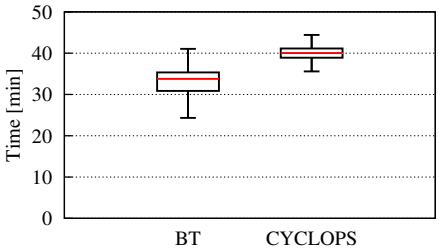



Fig. 4. **Waves of Arrivals**: content download times (file size: 50MB).

Table II shows the average volume of data served by both schemes, as well as information on traffic overhead (namely, volume of control messages involving off-cloud bandwidth resources). For CYCLOPS, we show the aggregate overhead incurred by the content server and the swarm monitor. For completeness, we report the feedback traffic exchanged between the content server and OF node, noting that these messages are exchanged within the confines of the cloud and hence do not entail additional costs. The data in Table II corroborates our conclusion that CYCLOPS achieves low off-cloud resource utilization, even when the system is artificially stressed by complex client dynamics.

TABLE II
**WAVES OF ARRIVALS**: SERVER LOAD & OVERHEAD (FILE SIZE: 50MB)

| | BT | CYCLOPS |
|---|---|---|
| Normalized server load | 39.86 | 1.5 |
| Outgoing overhead | 55 KB | 52 KB |
| Incoming overhead | 2560 KB | 716 KB |
| Feedback overhead | – | 145 KB |

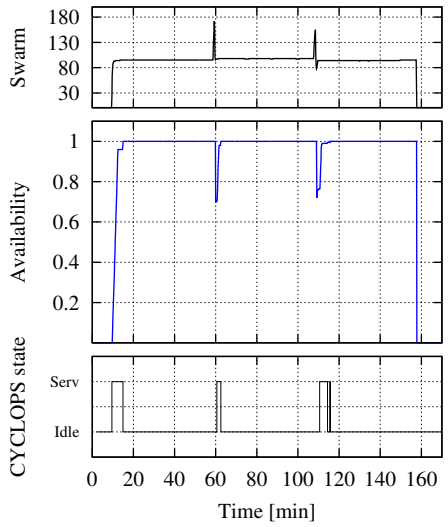



Fig. 5. **Waves of arrivals**: availability over time.

Next we examine the evolution in time of the feedback signal (namely, system-wide piece availability) generated by the CYCLOPS swarm monitor and the content server state transitions it triggers. Let $M$ be the number of pieces into which a file is divided, and let $I(i,t)$, $i = 1,\ldots,M$ be the indicator function for piece $i$ at time $t$, *i.e.*, $I(i,t) = 1$ if there is at least one copy of piece $i$ at time $t$, otherwise $I(i,t) = 0$. The availability feedback signal $A(t)$ at time $t$ is computed as:

$$A(t) = \frac{\sum_i I(i,t)}{M} \tag{4}$$

Figure 5 shows the time-series for the swarm size, the availability feedback signal, and the content server state transitions induced by this signal. It shows that as soon as the feedback signal indicates piece starvation (*i.e.*, availability is less than 1), the content server switches to the serving state and feeds the swarm. Piece availability is zero when the swarm bootstraps, and drops whenever clients holding the unique copy of a particular piece depart from the system. The content server switches from the idle state to the serving state only when necessary to restore piece availability to 1. Note that in this experiment we have purposefully created an extreme case of swarm dynamics: in a real swarm, user behavior is not as synchronous.

### *C. Live Internet Experiments*

In the set of experiments we present in this Section, we do not control the client arrival and departure processes, but rather

we let these processes reflect the popularity of the content we advertised. Furthermore, clients participating in our swarms exhibit realistic uplink and downlink capacities, unlike our PlanetLab experiments in which all clients have the same uplink capacity.

For CYCLOPS, out of a total of 7633 users we tracked, 3509 obtained the full content. All other users departed before finishing the download process. For the open-loop-BT content server, 2486 out of a total of 5044 users completed the content download. Figure 6 depicts the instantaneous number of users for both swarms.[12]

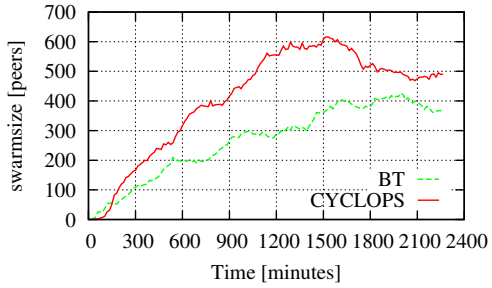



Fig. 6. **Live Experiment:** Evolution of swarm size over time.

Figure 7 shows the box-plot of the content delivery times achieved by all users that were able to complete the download. These results indicate that the median delivery time achieved by both content servers is very similar. For the CYCLOPS content server, the ECDF indicates longer tails: this is mainly due to a larger swarm size, which included clients with poor Internet connectivity. From the end-users' perspective, the difference in the download performance when they are served by CYCLOPS or by open-loop-BT is negligible.

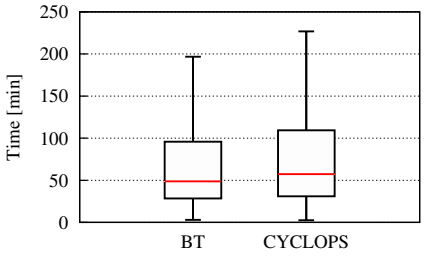



Fig. 7. **Live Experiment:** content download times (file size: 357.5 MB).

The off-cloud bandwidth utilization, the associated volume of data and related costs supported by content servers underscore the superiority of CYCLOPS. Table III indicates that the CYCLOPS content server served a total of 731.6 MB of content data, while the open-loop-BT seed injected a whopping 133.03 GB of content data! Table III also reports the overhead traffic, as defined in the previous section.

These results[13] support our conclusion that the framework discussed in Section II and the particular instance we presented in this work are viable candidates for real Internet content distribution systems. Since we deployed our content servers on Amazon EC2 instances, we were able to quantify the economic value of our proposed scheme: For the experiment we carried out, the total cost (including overheads) for distributing the same content when using a legacy BT seed is roughly 180 times higher that of a CYCLOPS content server.

TABLE III
**LIVE EXPERIMENT:** SERVICE STATISTICS (FILE SIZE: 357.5 MB)

| | BT | CYCLOPS |
|---|---|---|
| *Total number of users observed in the swarm* | 5044 | 7633 |
| Normalized server load | 381.04 | 2.05 |
| Outgoing overhead | 6.5 MB | 0.2 MB |
| Incoming overhead | 160.8 MB | 24.6 MB |
| **Cost of delivery** | $ 23.73 | $ 0.13 |

## VI. ADDITIONAL CONSIDERATIONS

We now discuss several points that complement the work presented in this paper. We start by suggesting practical ideas to implement a content server with alternative objectives and feedback signals; then we address the case for multiple content servers and conclude with a discussion of the robustness of our framework against attackers aiming at thwarting the content distribution process.

**Dealing with alternative objectives and feedback signals:** The framework proposed in Section II is general enough to allow many possible combinations of objectives and feedback signals. For example, an alternative objective may be to ensure some minimal level of service based on a feedback signal regarding the *average* delivery time of content to clients. The swarm monitor described in Section III can readily measure the average content delivery times, using the same swarm signaling traffic we discussed earlier. Indeed, clients advertise whenever they receive a new content piece, information that can be simply used to compute the average download rate of the swarm. Based on this information, the content server can choose the appropriate level of off-cloud bandwidth (*i.e.*, the cost it incurs) to complement the serving capacity $\lambda$ of the swarm, with the constraint of remaining in the vicinity of the quiescent operating point discussed in Section II. With reference to Figure 1, this approach corresponds to a content server selecting to contribute bandwidth resources that move across the various operating regions obtained for different values of $\lambda$.

**Dealing with alternative ways to collect feedback signals:** The swarm monitor described in Section III is achieved using a set of OF nodes that connect to all users. We show in Section V that the cost of this solution, in terms of overheads, is not significant. Nevertheless, maintaining many connections may pose some challenges. An alternative solution is to use periodic sampling of the swarm state: The OF nodes, instead of

[12]In our experiments, after the transients of the first few hours have subsided, the user arrival and departure rates within each swarm equalized, with approximately 35-40 users joining each swarm per minute.

[13]Note that both experiments lasted 38 hours, and that the swarm sizes allowed us to assume equivalent uplink capacity distributions for users in each torrent.

connecting to all the users in the swarm, periodically obtain a subset of users from the tracker and connect temporarily to this subset to collect the information about pieces owned by the users. Using sampling statistics, it is possible to *infer* system-wide piece availability, subject to preset levels of confidence. Clearly, the larger the sampling set, the more precise the availability information: in practice, approximating data availability may yield higher server load, since pieces may not be detected even if they are in the swarm.

**Dealing with multiple content servers:** In this paper, we conducted experiments in which a single content server is deployed. There are many obvious reasons to consider a more general scenario involving multiple content servers. For example, a CDN operator may wish to use CYCLOPS on edge servers positioned in several locations so as to serve clients efficiently: in this scenario, end-users might be directed to their geographically closest CYCLOPS content server. Traffic locality to mitigate the impact on ISPs economics, calls for a technique to create distinct swarms. This can be achieved with techniques proposed in the literature without requiring any modification to the design of CYCLOPS. Alternatively, multiple CYCLOPS servers could be combined to contribute to the same swarm. In this case, such content servers would have to coordinate what content pieces they serve and when to avoid inefficiencies. Our current implementation does not have provisions for avoiding the overlap between the *serving sets* compiled by different content servers. That said, standard distributed algorithms could be easily used to manage such situations for production-scale systems.

**Dealing with adversarial workloads:** Denial of Service attacks as well as other improper behavior of end-users aiming to exploit swarm resources is a concern that has to be considered when embracing a peer-assisted CDN solution such as ours. Although this is an important problem to address, here we focus on deliberate attacks by a client (or a set of colluding clients) targeting the specifics of our CYCLOPS framework.[14] We recognize two possible adversarial exploits, where the aim is to pollute the feedback signal computed by the CYCLOPS swarm monitor.

In the first, an adversary may seek to consume as much off-cloud bandwidth as possible. This can be done by inducing the content server to detect piece starvation (when none truly exists), thus causing the server to wastefully inject content. Since CYCLOPS swarm monitor tracks *all* clients in a swarm, such an attack would require a colluding set of malicious users of a size approximately equal to the whole swarm size, which can be safely assumed impractical.

In the second, a set of colluding users may engage in a DoS-like attack to hinder content distribution, by inducing the content server to conclude that the swarm is healthy (when the contrary is true). This causes starvation of legitimate clients. This can be solved by letting the swarm monitor to compute the average download rate of the swarm (as explained before in this Section). Based on this information, in case of content starvation, the swarm monitor may trigger an alarm, indicating, for instance, the less replicated pieces.

[14]Other types of attacks typical of P2P systems, such as Sybil or Eclipse attacks, can be solved using the techniques already presented in the literature[20].

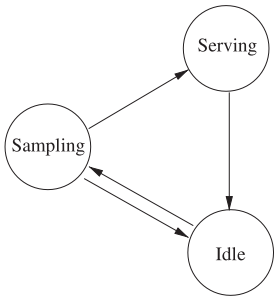



Fig. 8. The Sampling-Serving-Idle scheme, with transitions among different states.

## VII. APPROACH SIMPLIFICATION

We now discuss a simplification of our approach to collect swarm information. So far, we used continuous feedback signals provided by the constant monitoring of OF nodes. To simplify our architecture, we develop some of the considerations in VI, preferring to the continuous monitoring a periodic sampling strategy to check the swarm status. Using this approach, there is no need of keeping several connections open all the time, thus the system needs less resources, sparing us from the deployment of the OF nodes. Indeed, in this case the content server can absolve the OF nodes from the swarm monitoring, adding to the serving and idle states a new *sampling* state.

We denominate this content delivery policy as the Sampling-Serving-Idle (SSI) scheme. The Content server can be in one of the following three states: Sampling, Serving or Idle. In the Sampling state, the server samples the swarm status. In the Serving state the server injects required pieces. In the Idle state the server does not upload content. Next, we describe each state in detail.

**Sampling:** In the Sampling state the content server interacts with a small set of peers, labelled the *sampling set*, currently downloading the content, and infers an instantaneous measure of content availability using a set of control messages specified in the application protocol. With this information at hand, the content server builds a *serving set*, i.e., the set of missing pieces in the swarm. If the serving set is empty, the content server switches to the Idle state, otherwise the Serving phase is triggered.

In our BT compatible implementation the server collects “have” and “bitfield” messages from a sampling set of 50 peers. The sampling set is obtained from the tracker each time the content server enters in the Sampling state. Clearly, the larger the sampling set, the more precise the availability information: in practice, approximating data availability by defect may yield higher server load, since pieces may not be detected (and hence inserted in the serving set) even if they are in the swarm.

**Serving:** This state is identical to the *serving* state of CYCLOPScontent server described in III. When the server has finished to upload all missing pieces, it goes to the Idle state. **Idle:** here the content server simply closes all the connections to remote peers and refuses any incoming connection. A server remains in the Idle state for $T_{\text{Idle}}$ seconds, after which a Sampling phase begins. The frequency at which the server uploads missing pieces clearly depends on the time spent in the Idle state ($T_{\text{Idle}}$), since during the Serving phase each piece is served at most once. Hence, $T_{\text{Idle}}$ and the upload bandwidth of the content server determine the volume of data injected in the swarm.

We now describe a *baseline* content serving policy that adapts the volume of data served by the content source by updating the value of $T_{\text{Idle}}$, which is achieved without the need of any additional information on the system state. We set a bootstrap value to the time spent in idle state to $T_{\text{Idle}}^{\text{min}}$. We then adopt a multiplicative increase, multiplicative decrease (MIMD) approach in updating $T_{\text{Idle}}$. Each time the content server switches from the Sampling to the Idle state, $T_{\text{Idle}}$ is multiplied by a factor of 2; the idle time cannot increase above $T_{\text{Idle}}^{\text{max}}$. Each time the content server switches from the Serving to the Idle state, $T_{\text{Idle}}$ is multiplied by a factor of 0.5. Intuitively, as data availability is at risk, the content server increases the sampling frequency. Instead, the more a swarm appears to be in a "healthy" state, the less frequent the content server is intervene. Such approach allows the content server to adjust dynamically its sampling rate, offering a way to spare greatly its resources compared to a fixed timing sampling approach.

### A. A comparison of CYCLOPS and SSI

We repeated the same experiments described in IV to compare the CYCLOPS and SSI performance.

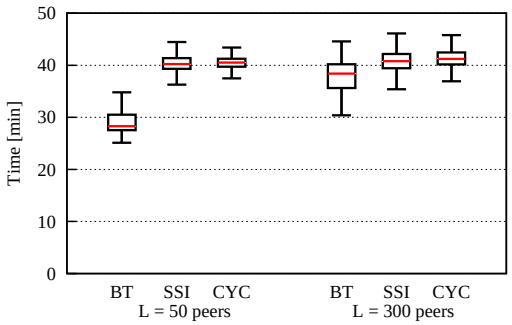



Fig. 9. **Flash Crowd**: content download times (file size: 50 MB).

Fig. 9 shows the download time in the flash crowd experiment. We note the perfomance of both SSI and CYCLOPS are comparable. The only notable difference is shown in Table IV: the SSIcontent server has a slightly higher load w.r.t. the CYCLOPS one. That is due to the accuracy of the estimate of piece availability. Indeed, in SSI, since the seed samples a subset of 50 peers only, it may happen that the availability of some pieces, especially those recently uploaded, is not detected: as a consequence, the seed re-injects them. In CYCLOPS case, the knowledge of piece availability is more precise: in this static scenario, it is not necessary to upload more than one copy of the content.

TABLE IV
**FLASH CROWD**: AVERAGE SERVER LOAD (NORMALIZED TO CONTENT SIZE: 50 MB)

| | BT | SSI | CYCLOPS |
|---|---|---|---|
| $L = 50$ | 12.2 | 1.16 | 1 |
| $L = 300$ | 15.36 | 1.96 | 1 |

Also Fig. 10 indicates that perfomance of both SSI and CYCLOPS are comparable when considering the download time as metric. Again the the server load table (Table V) reveals an important difference in load between SSI and CYCLOPS. Note also that the SSI case shows a lower incoming overhead when compared to CYCLOPS because the latter monitors continuously more nodes: thus, this parameter grows with the swarm size. However, the signaling traffic between OF nodes and the content server is negligible.

Plotting the estimate, done by the SSI server, of the system-wide piece availability is helpful to understand the different load on both the two kind of content server.

Fig. 11 shows the results for the SSI scheme, where the availability plot includes both the estimate computed by an OF node (continuous line) and the one computed by the SSI content server (the dashed line). Note that the OF node does not supply any information to the content server in this case. As long as the availability from the seed viewpoint is less than 1, the seed injects pieces in the swarm. When the availability equals 1, the Sampling intervals follow a

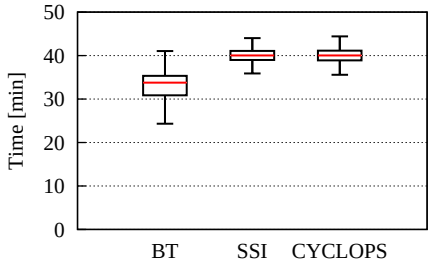
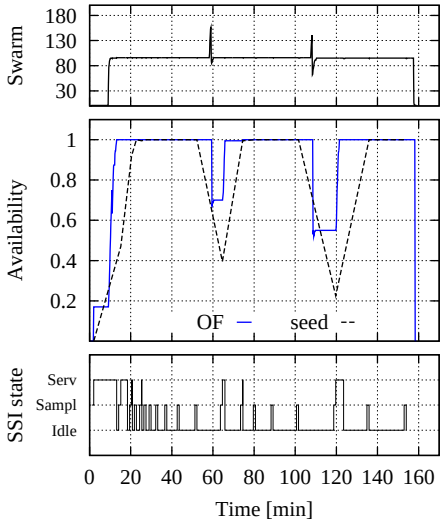



Fig. 10. **Waves of arrivals**: content download times (file size: 50 MB).

TABLE V
**WAVES OF ARRIVALS**: AVERAGE OVERHEAD AND SERVER LOAD (CONTENT SIZE: 50 MB)

| | BT | SSI | CYCLOPS |
|---|---|---|---|
| Normalized server load | 39.86 | 2.08 | 1.5 |
| Outgoing overhead | 55 KB | 52 KB | 52 KB |
| Incoming overhead | 2560 KB | 212 KB | 716 KB |
| Feedback overhead | – | – | 145 KB |

Fig. 11. **Waves of arrivals**: availability over time in SSI.




multiplicative increase pattern: transitions to the Sampling state are less and less frequent. As soon as the availability drops due to peer departures, the seed starts serving again. Clearly, underestimating piece availability triggers frequent transitions to the Serving state, even when it is not necessary.

In summary, the SSI policy ensures long-term piece availability only: this approach is less reactive to a highly dynamic swarm. Instead, CYCLOPS constantly tracks piece-availability and can respond promptly to peer dynamics: this feature comes at the cost of an increased overhead due to OF nodes. Moreover, on the one end SSI reduces the complexity of system deployment, while CYCLOPS requires a set of OF nodes to work. On the other hand, the CYCLOPS parameters which regulate the SSI transition must be tuned in order to adapt the server reactivity to the users behavior in a particular content distribution scenario. Tuning these parameters per each content is a difficult and tedious operation that evidence the practical advantage of adopting CYCLOPS over SSI.

## VIII. RELATED WORK

**Peer-Assistance:** Peer assisted content distribution have been the subject of many recent studies. Of these, the work of Huang, Wang, and Ross [12] could be seen as similar in nature to the work presented in this paper. In that work, the authors advocate the use of peer-assisted content distribution by evaluating the potential gain from peer-assisted video distribution using real-world traces of two large CDN companies, Akamai and Limelight (the underlying architecture of both of which they characterized). Their approach uses the model in [11] to obtain bounds on the server load and download times, should swarming among end-users be allowed. They also quantify the potential reduction in ISP peering traffic, resulting from traffic localization. In the same vein, our work is based on an analytical model that gives key insights as to the benefits of peer-assisted content distribution (although, our focus is on bulk as opposed to video transfers). Beyond a "proof of concept" using a tractable mathematical formulation, we go one step further by presenting practical feedback-control content injection policies that aim to satisfy performance objectives while minimizing provider's costs. Our implementation is evaluated in realistic contexts, and our results go beyond a purely theoretic estimation of the benefits of peer-assisted content distribution.

**Frugal Seeding:** To the best of our knowledge, the only work that has a similar objective to ours – in terms of reducing the load/cost on a content source, albeit in a very different setting – is Sanderson and Zappala's work [19]. In that work, once the seed has determined a subset of pieces that should be injected in a swarm, it will satisfy any number of requests for those pieces. As a consequence, their technique does not offer the same level of control on the seed workload as the policies we study in this work. Indeed, we observe that for experiments carried out in similar settings, our content servers inject orders of magnitude less traffic than what was documented in [19]. Additionally, our system does not require any parameter to be empirically set.

Chen *et al.* [8] study the "SuperSeeding" mode introduced by an alternative BT client to help peers with slow Internet connections perform initial content seeding. The objectives of "SuperSeeding" are different from ours. Moreover, a number of problems due to multiple peers using "SuperSeeding" have been reported. The work in [6] proposes a "Smartseed" policy, which advocates serving just one copy of each piece. Besides the fact that Smartseed does not take into account dynamic scenarios, it requires the modification of clients, while our system involves changes only to the server with no modification to the client.

**Models and Bounds:** The literature is rich with analytical models that dissect many aspects of P2P content distribution. In [14] and [21], the authors derive lower bounds for the minimum content distribution time of a swarm-based P2P application: we build upon those works, but focus instead on the relation between the content server upload rate and the download rate achieved by peers. The work in [18] belongs to the family of fluid models of BitTorrent-like applications: however, in this model it is the number of peers (as opposed to traffic) in the system that is taken as fluid. The authors in [18] develop a differential equation for the fluid model, from which they determine the performance of the dynamic system. We also model content replication in a dynamic setting, but instead consider the number of piece replicas as the dynamic variable modeled using a Markov process.

**Bandwidth Allocation in P2P Systems:** While the study of alternative mechanisms that improve the bandwidth allocation in P2P systems is orthogonal to our work, results from such studies could clearly have positive implications on content server utilization. In [17], the authors design a content distribution system with the objective of maximizing the download rate of all participants in a managed swarm. This work stems from the observation that, in steady state, a swarm can be in three different states: if the upload bandwidth allocated by content servers is insufficient, peers will not be able to fill

their uplink capacity and the aggregate download rate will suffer; by increasing the amount of bandwidth awarded to a single swarm, the content server can guide the system to operate in a regime where the uplink capacity of peers is gradually filled, up to a point in which also the downlink capacity of all peers is filled; at this point, server capacity can be diverted to other swarms. The system design in [17] is based on a wire protocol that induces peer participation (using virtual currency) to achieve a global system optimization. In our work, we focus on a different objective: we try and address the question of whether it is possible to optimize the bandwidth utilization by content servers, without negatively impacting the performance perceived by clients. We note that the model we use in this work can also explain, though in more general terms, the key intuition behind the Antfarm work [17].

The problem of devising efficient uplink allocation algorithms for swarm-based P2P bulk data transfers is addressed in [15]. Instead of using empirically set parameters, as done in BT, to determine the amount of uplink capacity dedicated to each remote connection, they cast uplink allocation as a fractional knapsack problem, and design a simple heuristic utility function to decide the amount of bandwidth a peer should dedicate to each remote connection. The focus of their work is on a cooperative P2P setting, in which peers are assumed to fully abide to the prescribed algorithms.

## IX. Conclusion

In this paper, we have demonstrated that peer-assisted content distribution could be leveraged to *supplant* as opposed to *supplement* the content provider's resources for purposes of efficient and scalable content distribution, *without* negatively impacting the performance perceived by clients. Our approach is based on a feedback-controlled swarm feeding mechanism, which we have modeled analytically and evaluated empirically using CYCLOPS – a full-fledged service that we have implemented and deployed on the Amazon EC2 cloud.

Our extensive experimental results – including the *live* distribution of content to thousands of real Internet users – show that CYCLOPS achieves enormous cost savings for the provider (as high as two orders of magnitude when compared to non-feedback-controlled BitTorrent-based services) without noticeably impacting the performance perceived by end-users. By deploying our servers on Amazon EC2 servers we were able to show that the mechanisms we developed as part of this work have a clear impact on content distribution economics, including significant reduction of costs for content providers, and much more efficient resource utilization for content hosts and distributors.

Our on-going work is focused on exploring alternative objectives and alternative feedback signaling processes in CYCLOPS, as well as extensions that take into account multiple (possibly competing) content servers involved in the distribution of content from multiple sources.

**Acknowledgments:** This research was supported in part by NSF awards #0720604, #0735974, #0820138, and #0952145.

## References

[1] http://aws.amazon.com.
[2] http://en.wikipedia.org/wiki/BitTorrent_(protocol).
[3] http://www.akamai.com.
[4] http://www.bittorrent.com.
[5] http://www.limelightnetworks.com.
[6] A. R. Bharambe, C. Herley, and V. N. Padmanabhan. Analyzing and improving a bittorrent networks performance mechanisms. In *Proc. of IEEE INFOCOM*, 2006.
[7] F. Bin, D.-M. Chiu, and J. C.S. Lui. Stochastic analysis and file availability enhancement for bt-like file sharing systems. In *Proc. of IEEE IWQoS*, 2006.
[8] Z. Chen, Y. Chen, C. Lin, V. Nivargi, and P. Cao. Experimental analysis of super-seeding in bittorrent. In *Proc. of IEEE ICC*, 2008.
[9] D. R. Choffnes and F. E. Bustamante. Taming the torrent: A practical approach to reducing cross-isp traffic in p2p systems. In *Proc. of ACM SIGCOMM*, 2008.
[10] R. Cuevas, N. Laoutaris, X. Yang, G. Siganos, and P. Rodriguez. Deep diving into bittorrent locality. Technical report, arxiv.org/abs/0907.3874, Telefonica Research, 2009.
[11] C. Huang, J. Li, and K.W. Ross. Can internet vod be profitable? In *Proc. of ACM SIGCOMM*, 2007.
[12] C. Huang, J. Li, A. Wang, and K.W. Ross. Understanding hybrid cdn-p2p: Why limelight needs its own red swoosh. In *Proc. of ACM NOSSDAV*, 2008.
[13] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, 1997.
[14] R. Kumar and K.W. Ross. Optimal Peer-Assisted File Distribution: Single and Multi-Class Problems. In *Proc. of IEEE HOTWEB*, 2006.
[15] N. Laoutaris, D. Carra, and P. Michardi. Uplink allocation beyond choke/unchoke or how to divide and conquer best. In *Proc. of ACM CONEXT*, 2008.
[16] A. Legout, G. Urvoy-Keller, and P. Michiardi. Rarest first and choke are enough. In *Proc. of ACM IMC*, 2006.
[17] R. S. Peterson and E. G. Sirer. Antfarm: Efficient content distribution with managed swarms. In *Proc. of USENIX NSDI*, 2009.
[18] D. Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *Proc. of ACM SIGCOMM*, 2004.
[19] B. Sanderson and D. Zappala. Reducing source load in bittorrent. In *Proc. of IEEE ICCCN*, 2009.
[20] M. Steiner, E. W. Biersack, and T. En-Najjary. Exploiting kad: Possible uses and misuses. *Computer Communication Review*, 37(5), 2007.
[21] R. Sweha, A. Bestavros, and J. Byers. Angels – in-network support for minimum distribution time in p2p overlays. Technical Report BUCS-TR-2009-003, Boston University, 2009.