

2017-10-01

# AngelCast: cloud-based peer-assisted live streaming using optimized multi-tree constructi

---

Vatche Ishakian, Raymond Sweha, Azer Bestavros. 2017. "AngelCast: cloud-based peer-assisted live streaming using optimized multi-tree construction." *Computer communications*, v. 111, pp. 14 - 28 (15).

<https://hdl.handle.net/2144/25732>

*"Downloaded from OpenBU. Boston University's institutional repository."*

# AngelCast: Peer-Assisted Live Streaming Using Optimized Multi-Tree Construction

Vatche Ishakian<sup>a,\*</sup>, Raymond Sweha<sup>b</sup>, Azer Bestavros<sup>c</sup>

<sup>a</sup>*Bentley University, CIS Department, Waltham, MA*

<sup>b</sup>*FreeWheel, New York, NY*

<sup>c</sup>*Boston University, Computer Science Department, Boston, MA*

---

## Abstract

Increasingly, commercial content providers (CPs) offer streaming solutions using peer-to-peer (P2P) architectures, which promises significant scalability by leveraging clients' upstream capacity. A major limitation of P2P live streaming is that playout rates are constrained by clients' upstream capacities – typically much lower than downstream capacities – which limit the quality of the delivered stream. To leverage P2P architectures without sacrificing quality, CPs must commit additional resources to complement clients' resources. In this work, we propose a cloud-based service *AngelCast* that enables CPs to complement P2P streaming. By subscribing to *AngelCast*, a CP is able to deploy extra resources (angel), on-demand from the cloud, to maintain a desirable stream quality. Angels do not download the whole stream, nor are they in possession of it. Rather, angels only relay the minimal fraction of the stream necessary to achieve the desired quality. We provide a lower bound on the minimum angel capacity needed to maintain a desired client bit-rate, and develop a fluid model construction to achieve it. Realizing the limitations of the fluid model construction, we design a practical multi-tree construction that captures the spirit of the optimal construction, and avoids its limitations. We present a prototype implementation of *AngelCast*, along with experimental results confirming the feasibility of our service.

---

<sup>☆</sup>Supported in part by NSF awards #0720604, #0735974, #0820138, #0952145, #1012798 #1012798 #1430145 #1414119.

\*Corresponding author. Tel: +17818912974

*Email addresses:* [vishakian@bentley.edu](mailto:vishakian@bentley.edu) (Vatche Ishakian), [rsweha@bu.edu](mailto:rsweha@bu.edu) (Raymond Sweha), [Best@bu.edu](mailto:Best@bu.edu) (Azer Bestavros)

## 1. Introduction

Streaming high-quality (HQ) video content over the Internet is becoming a standard expectation of clients, posing significantly different challenges for today’s content providers (CPs) such as Netflix, Hulu, or IPTV, compared to the challenges associated with the best-effort delivery of low-quality streaming through CPs such as YouTube and Facebook. For example, Netflix reported last year that it is delivering streams at rates between 1.7 Mbps and 3.8 Mbps [20].

To be able to deliver streams with a high bit-rate, content providers resort to Content Delivery Networks (CDNs) to deliver their content. Those CDNs, in turn, started to tap into client upload bandwidth through the use of P2P architectures to alleviate some of their own costs. Examples of peer-assisted content distribution systems include Akamai’s Netsession [2], Octoshape Infinite Edge [22], and BitTorrent DNA [3]. The problem with pure P2P architectures is that the current offerings by ISPs provide significantly higher download rate than upload rate. The persistent gap between the average downlink and the average uplink capacity of peers creates a gap between the clients expectation in terms of download rate and what the uplink capacities of their peers allows. This gap can be overlooked in typical P2P file sharing or VoD when some peers linger in the swarm after finishing downloading, thus allowing other clients to utilize them as seeders. In the case of live streaming, due to its real-time continuing nature, clients at any point in time will have significantly higher average download bandwidth than their average uplink capacity. To deal with this, proposed peer-assisted streaming systems rely on dedicated provider servers (or “seeders” in P2P jargon), which must download the live stream first before uploading it to clients.

In this paper we propose a cloud-based “live stream-acceleration” service, *AngelCast*. By subscribing to *AngelCast*, a CP is assured that its clients would be able to download the stream at the desired rate without interruptions, while maximally leveraging the benefits from P2P delivery. *AngelCast* achieves this by (1) enlisting special servers from the cloud, called *angels*,<sup>1</sup>

---

<sup>1</sup>We introduced the notion of angels in [33], where angels were used for a different

which can supplement the gap between the average client uplink capacity and the desirable stream bit-rate. Angels are more efficient than seeders as they do not download the whole stream, but rather they download only the minimum fraction of the stream that enables them to fully utilize their upload bandwidth. In our architecture, the capacity that otherwise would have been wasted in downloading the full live stream to the servers can be channelled to help the clients directly. (2) Choreographing the connectivity between nodes (clients and angels) to form optimized end-system multi-trees for peers to exchange stream content, and (3) handling clients dynamic arrival and departure.

We present theoretical results that establish the minimum amount of angel capacity needed to allow all clients to download at a desirable rate, as a function of their downlink/uplink capacities. We show that this lower bound is tight by choreographing the connectivity of nodes in such a way that the optimal bound is achieved under a fluid model.

A good live streaming system would also minimize the start-up delay needed to assure continued service. We prove that the start-up delay is zero under the theoretical fluid model, but that in practical settings, the optimal construction leads to a start-up delay that is linear in the number of clients when relaxing the fluid model assumption. Moreover, an optimal construction may require building a full mesh between clients (with no bound on node degrees). These reasons lead us to develop a more practical approach that utilizes almost the minimal amount of angel capacity (predicted under the fluid model), while also ensuring that the start-up delay is logarithmic in the number of clients. Our practical approach relies on dividing the stream into substreams, each of which is disseminated along a separate tree. To download the stream, each client subscribes to all the substreams, whereas angels subscribe only to one substream, allowing them to upload at full capacity to clients while not wasting too much bandwidth in downloading the whole live stream. The idea of splitting the stream into substreams was proposed in prior work, most notably in SplitStream [4] and [17]. In the related work section, we discuss what we learned from those proposed techniques, and how we avoided some of their shortcomings.

Another limitation of the fluid (optimal) choreography is that it does not consider issues of churn due to node arrival and departures. We address

---

objective – namely to minimize the bulk download time for a fixed group of clients.

this limitation by incorporating membership management capabilities that ensure uninterrupted service with minimum startup delay. We achieve this by ensuring that the trees used in our construction are well balanced, and by avoiding degenerate cases. Our cloud-based service provides a *registrar* that collects information about clients, making fast membership management decisions that ensure smooth streaming.

We discuss the architecture of our proposed *AngelCast* system, and evaluate a prototype implementation against SopCast [30] – a commonly used P2P streaming client. The experimental results carried out on Emulab and PlanetLab show the utility of angels and the effectiveness of our choreographed live stream distribution.

The remainder of this paper is organized as follows: In Section 2, we present the theoretical model that bounds the minimum amount of angel upload bandwidth needed to deliver the stream to all clients with the required bit-rate. We also present an optimal fluid construction that achieves that bound and compute the start-up delay associated with it. We conclude that section by highlighting the effectiveness of using angels over using seeders for live streaming. In Section 3, we present our practical construction that avoids the impracticalities of the optimal construction by relaxing the fluid assumption and bounding the node degree. In Section 4, we present our *AngelCast* service architecture including the membership management techniques and the design of our protocol. In Section 5, we experimentally evaluate our *AngelCast* prototype against SopCast in Emulab and PlanetLab. In section 6, we review the related work. We conclude in Section 7 with a summary of results and directions for future research.

## 2. Theoretical Bounds

We adopt the Uplink Sharing Model (USM) presented by Munding in [19], wherein each client is defined solely by its upstream and downstream capacities. The client is free to divide its upstream/downstream capacity arbitrarily among the other nodes as long as the aggregate upload/download rates do not exceed the upstream/downstream capacity. The client uploads to all other nodes in a unicast fashion. Hereafter, we use the term *fluid model* to refer to the use of the Uplink Sharing Model along with the ability to infinitesimally divide link capacities. The provider  $P$  is the originator of the live stream, it has an upstream capacity of  $u(P)$ . The set of clients subscribed to the live stream is  $C$  of size  $c = |C|$ . Each client  $c_i \in C$  has an upstream

capacity of  $u(c_i)$  and downstream capacity of  $d(c_i)$ . We denote the clients aggregate upstream capacity by  $u(C) = \sum_{c_i \in C} u(c_i)$ . The aggregate angels' upstream capacity is  $u(A)$ , where  $A$  is a set of angels. We assume that the stream playout rate  $r$  is constant.<sup>2</sup>

Each client  $j$  should be able to download fresh live content with a rate  $x_j = \sum_{i \in C \cup A \cup P} x_{ij}$  greater than the playout rate  $r$ , where  $x_{ij}$  is the rate between nodes  $i$  and  $j$ . By definition the provider's upload bandwidth is not less than the playout rate  $u(P) \geq r$ , otherwise the provider cannot upload the live stream. Also, it is fair to assume that the downstream capacity of all clients is greater than the playout rate  $d(c_i) \geq r \quad \forall c_i \in C$ , otherwise these clients will not be able to play the live stream at the desirable playout rate.

### 2.1. Optimal Angel Allocation

In this subsection, we derive the minimum amount of angel upstream capacity needed in order for all clients to receive the live stream with rate  $r$ . First, we provide a lower bound on the angel upstream capacity, then find an optimal fluid allocation scheme achieving this bound.

**Theorem 1.** *The minimum angel upstream capacity needed for all clients to receive the stream at a prescribed playout rate  $r$  is:*

$$u(A) \geq \frac{c^2}{c-1} * (r - \frac{u(P)+u(C)}{c})$$

*Proof.* For a client to receive the stream live, its download rate should equal the playout rate. Thus, the slowest client should receive content with rate not less than  $r$ ;  $\min_{j \in C} \{x_j\} \geq r$ . Because the average is always greater than the minimum, the average download rate should exceed  $r$  as well; if  $\min_{j \in C} \{x_j\} \geq r$  then  $\frac{\sum_{j \in C} x_j}{c} \geq r$ .

First, let us consider the case of no angels. The uplink sharing model dictates that the aggregate downstream capacity cannot exceed the aggregate upstream capacity in the swarm:  $u(P) + u(C) \geq \sum_{i \in C} x_i$ . To optimally utilize  $u(A)$  of the upstream capacity of the angels, an angel must download

---

<sup>2</sup>It is recommended to use CBR for live streaming. But in the case of variable bit-rate encoding (VBR), we can use the optimal smoothing technique to achieve a constant bit-rate (CBR) [29].

fresh data with a rate of at least  $u(A)/c$  then upload it to all  $c$  clients. Thus, in case of using angels, we have:

$$\begin{aligned} u(P) + u(C) + u(A) &\geq \sum_{\forall i \in C} x_i + \frac{u(A)}{c} \\ \frac{u(P) + u(C) + u(A)}{c} - \frac{u(A)}{c^2} &\geq \frac{\sum_{\forall i \in C} x_i}{c} \geq r \end{aligned}$$

Rearranging this inequality allows us to derive the angel upstream capacity needed to achieve the prescribed playout rate  $r$ .

$$u(A) \geq \left(\frac{c^2}{c-1}\right) * \left(r - \frac{u(P) + u(C)}{c}\right)$$

□

Not surprisingly, the bound in Theorem 1 suggests that the capacity of needed angels grows linearly with the number of clients and with the deficit between the playout rate and the client share of the provider and clients upstream capacities.

**Theorem 2.** *All clients can achieve the playout rate  $r$  when:*

$$u(A) = \frac{c^2}{c-1} * \left(r - \frac{u(P)+u(C)}{c}\right)$$

*Proof.* We prove that the lower bound on the minimum angel upstream capacity is achievable by construction. Using a fluid model, we choreograph the transfer rates between nodes so as to achieve a download rate that equals the playout rate for all clients. The set of Equations 1 has these rates. The provider sends data to client  $c_i$  with rate  $x_{Pi}$ . The client  $c_i$  in turn forwards this data to other clients  $j \in C$  with rate  $x_{ij}$ . The provider sends data to the angel with rate  $x_{PA}$ , the angel relays this data to the  $c$  clients immediately.

$$\begin{aligned} x_{Pi} &= \frac{u(c_i)}{c-1} + \delta \quad \forall c_i \in C \\ x_{PA} &= \frac{u(A)}{c} \\ x_{ij} &= \frac{u(c_i)}{c-1} \quad \forall c_i \in C, i \neq j \\ \text{where } \delta &= \frac{u(P)-r}{c-1} \geq 0 \end{aligned} \tag{1}$$

These rates guarantee that each client receives data at rate  $r$  without violating the upstream capacity constraint of any node. The aggregate download rate for client  $j$  (from all sources) will be

$$\begin{aligned}
x_j &= x_{Pj} + x_{Aj} + \sum_{i \in C, i \neq j} x_{ij} \\
&= \frac{u(c_j)}{c-1} + \delta + \frac{u(A)}{c} + \sum_{i \in C, i \neq j} \frac{u(c_i)}{c-1} \\
&= \frac{u(C)}{c-1} + \frac{u(P)-r}{c-1} + \frac{1}{c} * \frac{c^2}{c-1} * \left( r - \frac{u(P)+u(C)}{c} \right) \\
&= \frac{u(C)}{c-1} + \frac{u(P)-r}{c-1} + \frac{c}{c-1} * \left( r - \frac{u(P)+u(C)}{c} \right) \\
&= r
\end{aligned} \tag{2}$$

The upload rate of each client,  $i$ , will not exceed its upstream capacity as  $(c-1) * u(c_i)/(c-1) = u(c_i)$ . The same can be said about the angels:  $c * u(A)/c = u(A)$ . Also, the aggregate upload rate from the provider will not exceed its capacity:

$$\begin{aligned}
&x_{PA} + \sum_{j \in C} x_{Pj} \\
&= \sum_{j \in C} \left( \frac{u(c_j)}{c-1} + \delta \right) + \frac{1}{c} * \frac{c^2}{c-1} * \left( r - \frac{u(P)+u(C)}{c} \right) \\
&= \frac{u(C)}{c-1} + c * \left( \frac{u(P)-r}{c-1} \right) + \frac{c}{c-1} * \left( r - \frac{u(P)+u(C)}{c} \right) \\
&= u(P)
\end{aligned} \tag{3}$$

To ensure that each client receives non-duplicate data, the provider sends unique data to the angels. As for the clients, each client receives unique data with rate  $u(c_i)/(c-1)$  and the same data with rate  $\delta$  to all clients.  $\square$

Figure 1 illustrates two examples of the optimal construction. The left side of Figure 1 is an example with three clients whose upstream capacities are sufficient to achieve a playout rate of  $r$ , thus there is no need for angels. Each client splits its upstream capacity between the other two clients. The provider sends data to clients with a rate that equals half their upstream capacity plus  $\delta$ . The  $\delta$  part of the upload rate is identical, in terms of its content, to all clients, while the other part is unique, ensuring the uniqueness of data disseminated. On the right side of Figure 1 is an example where the upstream capacity of the provider and the two clients is not enough to secure a playout rate  $r$ . Thus, an angel is needed. Here, the angel downloads from the provider and uploads to the clients. Each client downloads from the provider and uploads some of what it receives from the provider to the other clients (and angels do not download from clients).

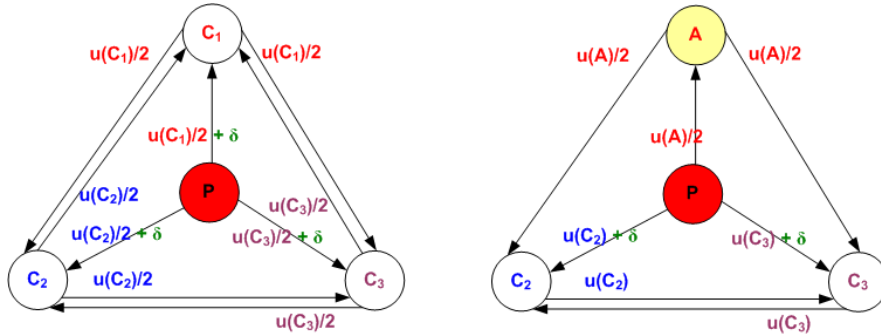


Figure 1: Illustrative examples of the optimal construction: 3 clients not in need of any angels (left) and 2 clients in need of one angel (right).

## 2.2. Implications on the Role of Angels

The premise behind this work is that there is a significant gap between the clients' upstream and downstream capacities offered by ISPs. The promised higher downstream capacity encourages content providers to stream at an ever-increasing rate. Pure P2P technology helps alleviate the cost on content providers by utilizing the clients upstream capacity. To bridge this gap, many peer-assisted file-based streaming constructions were proposed. Ours is the first to realize that it is more efficient for the added resources to download a fraction of the live stream instead of its entirety. Figure 2 illustrates the ecosystem of our angel-based content acceleration architecture. In this ecosystem, the aggregate downloaded data equals the aggregate uploaded data. The client upstream capacity cannot match the download rate of many streams. Thus, we need to add agents to this ecosystem that produce more than they consume. Angels provide that by downloading a fraction of the stream instead of consuming the already scarce resource of upstream capacity by downloading unnecessary content.

In live streaming, content providers do not have the luxury of uploading the content to many servers beforehand. Thus, those servers will compete over the upstream capacity with clients. In a peer-assisted streaming mechanism, like the one presented in [17], the system relies on more server capacity to stream at a higher stream rate not supported by the clients upstream capacity. Figure 3 shows a numerical analysis of the performance of angels against servers in a hypothetical setting. It compares the number of angels, each with upstream capacity ten, versus the number of servers, each with upstream capacity ten too, (on the y-axis) needed to achieve a streaming rate

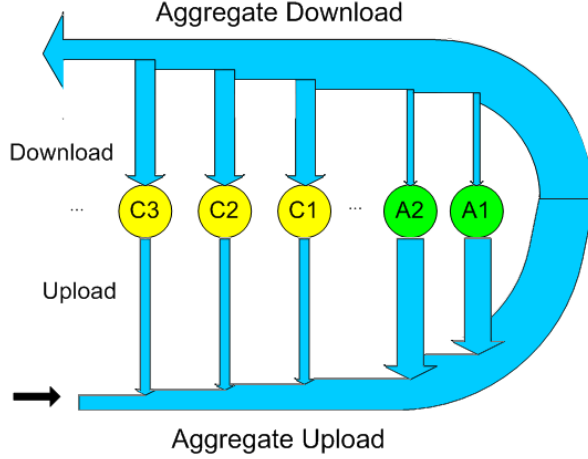


Figure 2: Our proposed ecosystem; clients download more than they can upload, and angels download as little as possible.

(on the x-axis). The results are shown for a family of curves with varying clients' ratios of upstream-to-playout rates – ranging from 3:10 to 9:10. The streaming rate (x-axis) varies between 1 and 9. There are 100 subscribed clients and the number of clients an angel can connect with concurrently is 40. The formula to compute how many servers we need is

$$\frac{c * r}{u(a) - r} * (1 - ratio)$$

and the formula for the number of angels is

$$\frac{c * r * k}{u(a)(k - 1)} * (1 - ratio)$$

The growth in the number of angels is linear with the stream rate. However, if we use servers, the growth is super-linear. For example, when the streaming rate is 9 and the server upstream capacity is 10, nine-tenth of a server capacity is wasted in uploading the stream to another server (90% overhead). The graphs compare different values for the ratio between the upstream capacity to the playout rate. The greater the gap, the more angels/servers the system will need. Clearly, angels are significantly more efficient than servers, especially when the stream playout rate is large.

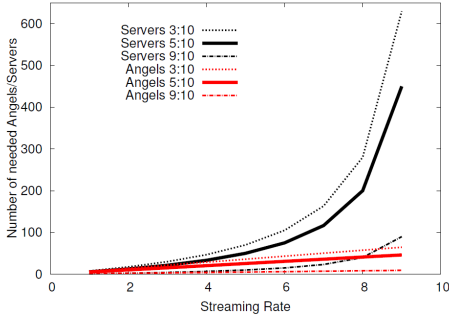


Figure 3: The number of angels versus the number of servers needed to achieve a desired playout rate for 100 clients.

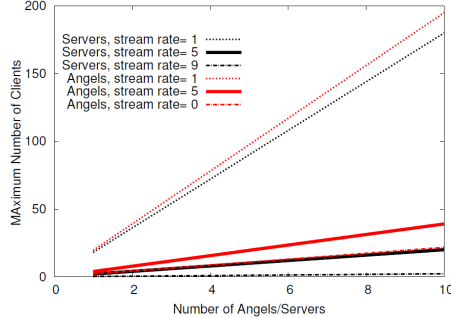


Figure 4: The number of clients that can be supported by a fixed number of angels or servers.

Figure 4 shows the maximum number of clients that can download at a desired rate given the number of available angels/servers. We fix the ratio between the upstream rate and stream playout rate to 1:2. At low stream rates, a few high-capacity angels/servers can support many clients. As the stream rate increases, the number of clients satisfied by the service decreases. Again, the angels are more efficient as they are able to serve more clients consuming the same amount or resources.

### 2.3. Startup Delay

In this section, we study the effect of packetization on startup delay. Assume that the unit of transfer is of size  $\psi$ . In the fluid model,  $\psi$  approaches zero. In a practical setting, each node cannot forward a packet to another node before it finishes receiving it.

**Theorem 3.** *Using the optimal construction in Theorem 2, the startup delay,  $D$ , is:*

$$D = \frac{\psi}{r} + \frac{\psi * (c-1)}{\min_{c_i \in C} u(c_i)}$$

*Proof.* The proof is by construction. The delay until all clients receive a certain packet consists of two parts; the first is the delay until a client receives this packet from the provider  $\psi/r$  and the second is the delay until it forwards the packet to all the other  $c - 1$  clients. The client that is forwarding the packet to all the other clients can be the one with the smallest upstream capacity, thus we divide the amount of data needed to be sent,  $\psi * (c - 1)$ , by the minimum upstream capacity  $\min_{c_i \in C} u(c_i)$ .  $\square$

Therefore, if the goal of the system is for all clients to enjoy an uninterrupted service, each client should fill a buffer of size  $B$  before starting the playout, where:

$$B = r * D = \psi + r * \frac{\psi * (c - 1)}{\min_{c_i \in C} u(c_i)}$$

This result suggests that the startup delay grows linearly with the packet size  $\psi$  and with the number of clients  $c$ . Under the fluid model assumption, this is not consequential because  $\psi = 0$ , resulting in a startup delay of zero as,  $\lim_{\psi \rightarrow 0} D = 0$ . In practical settings however,  $\psi \neq 0$  – *e.g.*, it could be the MTU of a TCP packet  $\simeq 1,400$  Bytes. In such cases, the optimal construction may result in significant startup delays for large number of clients. In the following section, we propose a dissemination strategy that achieves the desired download rate using minimum angels capacity while keeping the startup delay under a reasonable (logarithmic) bound.

### 3. A Practical Construction

The optimal construction in Theorem 2 requires each client to connect to all other clients. This leads to operational challenges and a start-up delay that grows linearly with the number of clients. To mitigate these problems we developed a new dissemination mechanism that constrains the node degree of each client to  $k$ , *i.e.*, each clients can communicate with at most  $k$  clients at any point in time. We call this construction *AngelCast*.

#### 3.1. Bounding the Angel Upstream Capacity

The limit on the node degree influences angel effectiveness inversely. Ideally, an angel would download a small fraction of the stream and would upload it to *all* clients. Under a bounded-degree assumption, it is intuitive that we would need more angel capacity. Theorem 4 provides a minimum lower bound on angel capacity needed.

**Theorem 4.** *The minimum angel upstream capacity needed for all the clients to download at the playout rate, when each node is constrained to connect to only  $k$  other nodes, is:*

$$u(A) \geq \frac{k*c}{k-1} * (r - \frac{u(P)+u(C)}{c})$$

*Proof.* Similar to the proof of Theorem 1, in order to optimally utilize  $u(A)$  of angel upstream capacity, an angel must download data at a rate of at least  $\frac{u(A)}{k}$  then upload it to a maximum of  $k$  clients, given that  $k \leq c$ . On the one hand, forwarding the same data to more than  $k$  clients *simultaneously* would violate the out-degree constraint, and on the other hand, forwarding the same data to more than  $k$  clients *on stages* would result in the reception of stale and out of stream data to some clients. Also, tearing down connections and building new ones frequently and systematically would result in performance degradation due to the nature of the transport protocols. To ensure that the aggregate upstream capacity is more than the aggregate download rate:

$$\begin{aligned}
 u(P) + u(C) + u(A) &\geq \sum_{\forall i \in C} x_i + \frac{u(A)}{k} \\
 \frac{u(P) + u(C) + u(A)}{c} - \frac{u(A)}{k * c} &\geq \frac{\sum_{\forall i \in C} x_i}{c} \geq r \\
 \text{Thus : } u(A) &\geq \left(\frac{k * c}{k - 1}\right) * \left(r - \frac{u(P) + u(C)}{c}\right)
 \end{aligned}$$

□

For small swarms, when  $c \leq k$ , the bounded-degree constraint is never reached, thus the bound on the angels upstream capacity – to other clients – does not change. Even when  $c > k$  and  $k$  is relatively large, we would not need significantly larger angel capacity as  $c/(c - 1) \simeq k/(k - 1) \simeq 1$ . For example if  $k = 100$  the overhead due to constraining the out-degree is around 1% even when the number of clients is extremely large.

### 3.2. Construction Under a Bounded-Degree Assumption

The optimal construction used in Theorem 2 assumes the ability of each client to connect to all other clients simultaneously. In the remainder of this section, we develop a practical construction under the constraint of limited node degree,  $k$ , where each node can communicate with only  $k$  other nodes at any point in time.

Our construction divides the stream into  $m$  substreams. We disseminate each substream using a multicast tree and each client subscribes to all the  $m$  trees. The rate of dissemination of all the substreams,  $r_t$  is equal, such that the sum of the substreams equals the playout rate, *i.e.*,  $r_t = r/m$ . This construction is similar to what is done in Splitstream [4] and CoopNet [24].

The number of trees,  $m$ , depends on the allowed degree of any node,  $k$ . For  $d$ -ary trees, each node is connected to at most  $d + 1$  other nodes (one parent and  $d$  children). Thus, the number of trees is bounded by:  $m \leq \lfloor k/(d + 1) \rfloor$ . Choosing the arity of the trees is not trivial. On the one hand, choosing a small arity allows for a greater number of trees, each with a small substream rate. This will minimize the unassigned client capacities  $\sum_{i \in C} (u(c_i) \bmod r_t)$  and utilizes angels more efficiently. As the theoretical results showed, angels are best utilized when they download the smallest substream which allows them to upload with their maximum bandwidth. On the other hand, choosing a larger arity would result in a smaller start-up delay, as we prove in Section 3.3. Also, choosing a large arity enables the utilization of the client with high upstream capacity in full because when a client subscribes to all trees as a parent of  $d$  children, it can forward data with maximum rate  $d * r_t * m = d * r$ . Thus, any client upstream capacity above  $d * r$  will be wasted. Consequently, choosing the right arity has to balance utilizing the resources and providing a small start-up delay.

Beside deciding on the number of trees and their arity, we need to decide on the placement of each client in each tree. First, let us consider the case in which there are no angels. Adding angels to this construction is straightforward and will be explained shortly. In our construction, each client calculates how many children it can adopt across all trees, which equals the upstream capacity of the node divided by the rate of a substream,  $r_t$ . Let us call this parameter  $l_i$ , where  $l_i = \lfloor u(c_i)/r_t \rfloor$ . Our construction dictates that these children be allocated in the minimum number of trees. This is necessary to avoid degenerate trees, where parents have only one child. Thus, the number of trees where this client can have  $d$  children is  $g_i = \lfloor l_i/d \rfloor$ . Client  $i$  can have the remaining children assigned to one more tree and it will be a leaf in all the other trees. Whenever a new client arrives, it will join all trees. To ensure that no tree is starving for bandwidth while another one has an abundance of it, the new client will be a parent in the trees where there are fewer places to adopt more children. We use *vacantSpots* to denote the number of places in a tree where it is possible to adopt more children.

The position of clients in each tree is equally important. To ensure a small start-up delay, the depth of the trees should be minimized. In a bounded-degree setting, the path from any node to the root should be logarithmic in the size of the swarm. Subsection 3.3 shows that full trees with large arity achieve that. Therefore, nodes that can adopt more children should be higher in the tree. Subsection 4.1 shows how we add/remove clients and

change connections while maintaining low-depth trees.

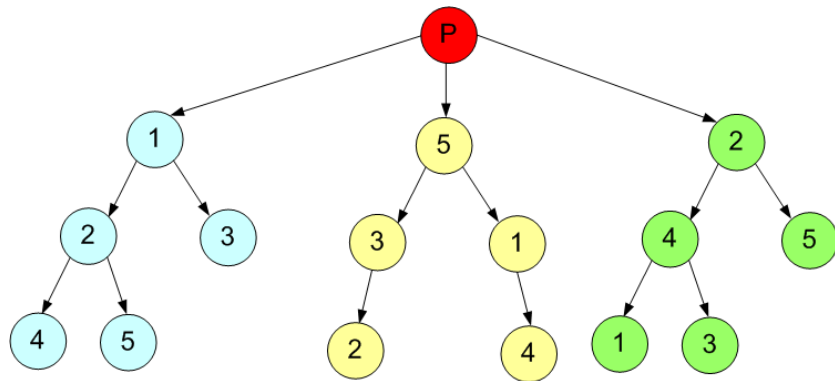


Figure 5: An example where the stream is split into three trees. Each node is a parent to as many children as possible in the least number of trees.

In all the trees, the order of a client in the tree is not important, as long as the clients who are assigned  $d$  children are in the top levels, the ones assigned less than  $d$  children are in the second to last level and the ones with no children are leaves. Figure 5 illustrates that. The number of nodes in the second to last level in any tree is at least one third of the number of the nodes in the tree. Thus if the number of trees is bigger than two, each client will be assigned to the second to last level at most once.

Adding angels to this construction is straightforward. The minimum upstream capacity of angels needed,  $u(A)$ , is given by the lower bound in Equation 4. This upstream capacity would be divided equally between a number of angels  $n_a = u(A)/(k * r_t)$ , each of which will be assigned to a different tree. Each angel will have  $k$  children in that tree. Whenever the number of vacantSpots in a tree falls below a certain threshold, we know that this tree is in poor health, hence we add an angel to that tree. When a tree has too many vacantSpots, we eliminate an angel, if any exists in this tree. Even though when an angel is added, it is added to a single tree, the health of the other trees will also improve. This is because newer clients will not need to become parents in this tree and will allocate their resources to other needy trees.

As for the provider, it will be a parent to the roots of the trees. The excess capacity of the provider can be utilized fully as well, as the provider can adopt more children. To avoid adding angels in all trees at the start, the

provider focuses its extra capacity in fewer trees.

To conclude, this construction allows each client to download with rate  $r$  and achieves near optimal utilization of the clients upstream capacity. The remaining upstream capacity that is not enough to adopt a child equals  $\sum_{i \in C} (u(c_i) \bmod r_t)$ , which can be seen as insurance in the case of bandwidth oscillations. The gap between the clients' upstream capacities and the payout (and hence download) rate can be supplemented by angels, and each node will not have to connect to more than  $k$  other nodes. In the following subsection, we show that our construction has, on average, a logarithmic startup delay in the number of clients,  $c$ .

### 3.3. Bounding the Startup Delay

In this section, we compute the startup delay given our construction in Subsection 3.2. A node with  $d$  children dedicates  $r_t$  of its upstream capacity to each one of them. Therefore, if we serialize the dissemination by sending a packet to a child at a time, the time to send a packet of size  $\psi$  to the first child will be  $\psi/(r_t * d)$ . The time to disseminate a packet of size  $\psi$  to all the  $d$  children, is  $d * \psi/(r_t * d) = \psi/r_t$  seconds. Each tree has  $c$  children, thus, it has  $\log_d(c)$  levels. Therefore, the time it takes for the last client in the last level of the tree to download a packet is:

$$D = \log_d(c) * \frac{\psi}{r_t} = \log_d(c) * \frac{\psi}{r * (d + 1)/k}$$

This startup delay is the minimum when  $\frac{\partial D}{\partial d} = 0$

$$\begin{aligned} \frac{\partial D}{\partial d} &= \frac{\psi * k * \ln(c)}{r} * \left( \frac{-\left(\frac{d+1}{d} + \log(d) * 1\right)}{\ln^2(d)} \right) \\ \ln(d^*) &= -\frac{d^* + 1}{d^*} \quad \text{At } \frac{\partial D}{\partial d} = 0 \\ d^* &= e \end{aligned} \tag{4}$$

The above means that in order to minimize the start-up delay, the degree of the tree should be maximized – *i.e.*,  $k - 1$ . This result illustrates the trade-off between minimizing the start-up delay and minimizing the needed angel capacity: The more trees we have, the better we are utilizing the angels/clients at the expense of increasing the start-up delay.

#### 4. AngelCast Architecture

Figure 6 shows the components in our peer-assisted service. While the registrar does not disseminate data, it choreographs the connectivity of clients and angels in the system. As indicated before, our service operates in a unicast setting. The registrar is the main agent in our cloud service. Content providers contact the registrar to enroll their streams. The registrar uses the profiler to estimate the upstream capacity of clients. The accountant uses the estimated gap between the clients' upstream capacity and the stream playout bit-rate to give the content provider an estimate of how many angels it will need.

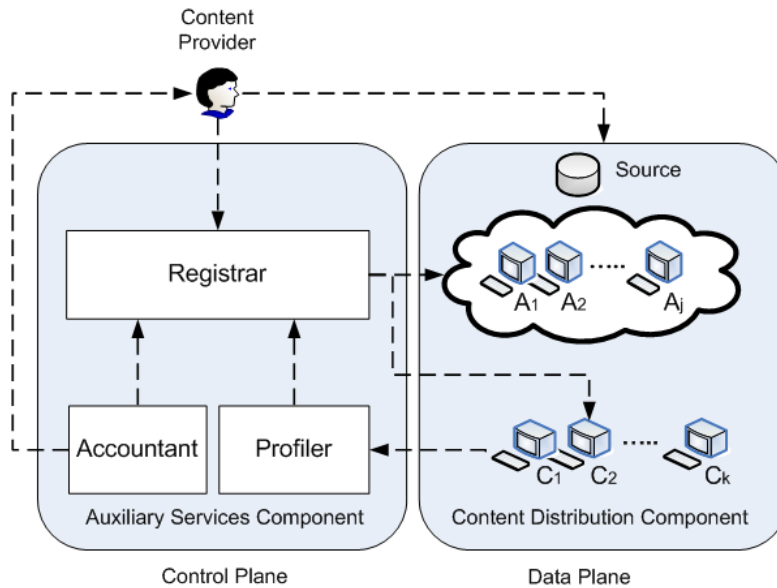


Figure 6: The architectural elements of *AngelCast*.

The *AngelCast* service could be incorporated and offered as a distinguishing feature by the cloud provider, or it can be implemented as a value-added proposition by a third-party service provider. The cost of the *AngelCast* services constitute an overhead that must be borne by the parties benefiting from the services (cloud provider, content provider, or clients). Clearly, there are many ways to appropriate the cost of such overhead. For instance, cloud providers could bear the cost of such service. Charging content providers or clients could also be an option. If clients are to collectively bear such costs,

one option may be to charge only the clients who use the service. Another option would be to distribute the cost in proportion to the benefit that clients get from the service. Yet, a third option would be to charge clients equally, independent of whether they benefit from the service or not.

---

**Algorithm 1** UpdateClosestAdopter()

---

```

oldAdopter = self.closestAdopter
if This node can adopt more children then
    self.closestAdopter = self
    self.closestAdopterDistance = 0
5: else
    minDistance = Infinity
    adopter = NULL
    for all child in ChildList do
        if child.closestAdopterDistance < minDistance then
10:         minDistance = child.closestAdopterDistance;
            adopter = child.closestAdopter
        end if
    end for
    self.closestAdopter = adopter
15: self.closestAdopterDistance = minDistance + 1
end if
if oldAdopter != self.closestAdopter then
    parent.UpdateClosestAdopter()
end if

```

---

#### 4.1. Membership Management: The Registrar

Live streaming swarms are dynamic in nature. Clients arrive and depart the stream constantly. Also, some bilateral connections between clients can degrade arbitrarily. Thus, it is absolutely essential to incorporate resilience in the design of swarms to ensure that some minimal quality of service is maintained. In this section, we explain how *AngelCast* handles membership management, *i.e.*, handling client arrival and departure, and replacing degraded connections with new ones. Our system relies on a special server to achieve that, the registrar. The registrar is a special node in the system that

orchestrates the swarm and choreographs the connectivity of the clients.<sup>3</sup> When a new node joins the stream, it contacts the registrar and informs it of its available upstream capacity. The registrar uses a data-structure representing the streaming trees and assigns the new client to a parent node in each tree. The registrar also decides how many future children the new node can adopt in each tree. Clients can also “complain” to the registrar about their parents. The registrar would pick a new parent for a complaining client, informs the client of its new parent, and also probes the under-performing parent to ensure it is still alive. If not, it pro-actively informs other children to disconnect from it and provides them with new parents.

These decisions are crucial in guaranteeing a continued service with low start-up delay and little disruption. In order to ensure fast response to clients’ requests, the registrar maintains a data-structure containing the state of each node in the system. The state of a node contains: (1) the depth of the subtree rooted at that node, (2) a pointer to the closest descendent with a vacant spot that can adopt one new child, which we call the *closestAdopter*, and (3) the distance to the closestAdopter. When a new client joins the swarm, the registrar adds it to the root’s closest adopter in each tree. The arrival and departure of clients changes this state. Algorithm 1 shows the function that updates the closestAdopter as well as the distance to it. The value of the closestAdopter can change for the new/old parent as well as for predecessors, recursively all the way to the root (by construction, a logarithmic process at worst). If the node has vacantSpots, then it is its own closestAdopter with distance zero. Otherwise, it picks the closestAdopter of one of its children, the one with the minimum distance to its closestAdopter. The update will propagate recursively along the path towards the root until the closestAdopter of a node along the path does not change.

However, this addition could change the root’s closestAdopter if it does not have vacantSpots for more children. In such a case, we will recursively update the closestAdopter value of the nodes in the path from the old closestAdopter to the root. If a node has more vacantSpots, it sets itself as the closestAdopter, otherwise, it embraces the closestAdopter of one of its children, the one with the minimum distance to its closestAdopter. By doing that, we are sure that each node is actually keeping track of its closestAdopter. The update will propagate recursively upward towards the root until

---

<sup>3</sup>Readers may observe some similarity between the registrar and a P2P system tracker.

the `closestAdopter` of a node along the path does not change.

As we alluded before in Subsection 3.3, in order to minimize the average start-up delay, we need to minimize the depth of the tree. We note that the degeneration of a tree could be caused by these few nodes that cannot have the maximum number of children. Our goal is to push these nodes down the tree to avoid this condition. In order to do so, we allow new nodes to *intercept* certain connections. By intercepting a connection we mean severing a connection between a parent and a *leaf* child node, making the parent adopt the new node, and making the new node adopt the child node. We prefer interception over adoption if the distance to the `closestIntercept` is smaller than the distance to the `closestAdopter`. We maintain and update the information about the *closestIntercept* and its distance for each node in the tree in a way similar to the `closestAdopter`. The pseudo-code for updating the `closestIntercept` is highlighted in algorithm 2.

The registrar receives complaints from nodes about their parents when they are not downloading at an adequate bit-rate. The registrar sends a probe to the parent, if the parent is alive and replies, the problem is with the link not with the parent. The registrar severs the connection to that parent and attaches the complaining child, and the subtree below it, to the `closestAdopter` in the tree. We need to ensure that the complaining node is not re-attached to the same parent, or worse to a descendent of its own. We ensure that by setting the `closestAdopter` distance of the parent to a very big number and propagating the update up the tree, forcing the root to choose a closest adopter away from the complaining node. We then attach the severed subtree to the new `closestAdopter` then set the `closestAdopter` distance of the old parent to zero and update up the tree. By the end of this process, the complaining node gets allocated, with its subtree, to a different part of the tree and the values of `closestAdopter` of all the nodes are adjusted.

Figure 7 illustrates how the value of the `closestAdopter` changes when a new client arrives. Before the addition, nodes C and D had `vacantSpots`. The root, node A, has node C as the `closestAdopter`. Thus node C will adopt the new node, F. This will change the value of the `closestAdopter` up the path to the root. Nodes C, F will have no `closestAdopter`. Thus, node A's `closestAdopter` will become node D, its other child, node B's, `closestAdopter`.

If the registrar's probe to a node results in no response, the registrar concludes that the client unsubscribed from the stream. Thus, it will actively remove it from all trees. When a node is removed from a tree, each of its orphaned children will be added, one by one, to the `closestAdopter` in the

---

**Algorithm 2** UpdateClosestIntercept()

---

```
oldIntercept = self.closestIntercept
if This node can adopt more children then
    self.closestIntercept = self
    self.closestInterceptDistance = 0
5: else
    minDistance = Infiniti
    intercept = NULL
    for all child in ChildList do
        if child is leaf then
10:         minDistance = -1
            intercept = self
        end if
        if child.closestInterceptDistance < minDistance then
            minDistance = child.closestInterceptDistance;
15:         intercept = child.closestIntercept
        end if
    end for
    self.closestIntercept = intercept
    self.closestInterceptDistance = minDistance + 1
20: end if
    if oldIntercept != self.closestIntercept then
        parent.UpdateClosestIntercept()
    end if
```

---

tree. To maintain balance in the tree, the children with smaller distance to their closestAdopter are added before the children with larger distance to their closestAdopter. Our service enables a graceful departure of clients by allowing them to declare their intention to leave the stream.

Figure 8 illustrates how our membership management techniques work through an example. Step 1 shows the initial state in which the provider is the root of the three trees, with an upstream capacity of 4 and a stream ployout rate of 3. Thus, the root,  $s$ , has 4 vacantSpots, two of which are assigned to the first tree and one vacantSpot for each of the second tree and the third tree. Client  $x$  joins in Step 2. It has two vacant spots, both of them will be assigned to the second tree. As we discussed before we assign a client as a parent in the minimum number of trees to maintain low depth

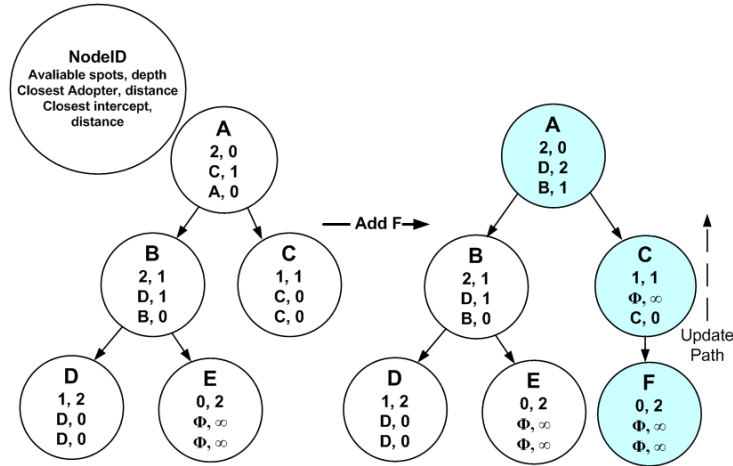


Figure 7: Node F joins the tree, the registrar updates the data structure accordingly.

trees. As a result of client  $x$ 's arrival, the number of vacantSpots in the third tree is reduced to zero. Thus a new angel,  $A$ , is added automatically to the third tree. The angel's upstream capacity equals 3, allowing it to have three children in one tree. If we had only one tree, the angel would have been useless as it will have had only one child and downloading as much as it is uploading. Because there are no vacantSpots in the third tree, the added angel will intercept the connection between the provider and client  $x$ . Step 3 illustrates the arrival of client  $y$ . Both of its vacantSpots will be assigned to the tree with the minimum number of vacantSpots, which is tree 1. In Step 4, client  $x$  complains about its connection to the provider in the first tree. The registrar disconnects it from the provider and instructs it to connect to client  $y$ . Step 5 illustrates the departure of client  $y$ . It will be removed from all trees and its children, if any, will be added one by one.

#### 4.2. The AngelCast Protocol

The registrar decides on the number of substreams, the fan-out of the trees and adds the provider as root to all trees. It also initializes the data structure in which the state of the system is kept. The registrar then starts a listener process to receive join requests and complaints from clients. It uses the membership management techniques, described in Subsection 4.1, to respond to such request. Whenever the number of vacantSpots in a tree falls below a certain threshold, the registrar instantiates a new machine from

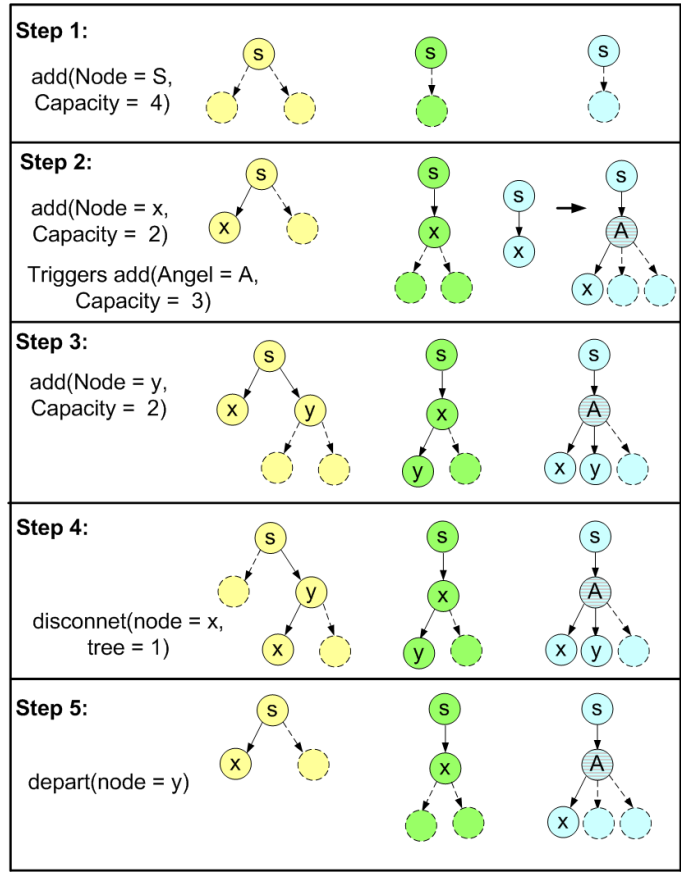


Figure 8: A hypothetical scenario illustrating the formation of *AngelCast* trees when clients join, leave or change parents.

the cloud as an angel. When a tree has too many vacantSpots, an angel is released from this tree, if any exists. A full implemented system, serving many live streams concurrently, can instantiate a couple of machines and leave them on standby at all time. Therefore, in the case when a stream is in need of help, an angel would be ready to help and there would be no need to wait for the typical delay associated with acquiring a machine from the cloud.

In our implementation, the provider and clients need to download plugins (software) to enable them to interact with the *AngelCast* system. On the provider side, the software divides live content as it arrives into substreams, which are maintained in separate substream buffers. The content of each

buffer is divided into chunks of fixed size. The provider also starts a listener process, which instantiates threads in response to join requests. These threads read from a substream buffer and sends chunks of data to the client. On the client side, the software starts by contacting the registrar asking to join a stream. The registrar replies with information about the stream as well as the identity of a parent capable of serving the (sub)stream in each tree. Upon contacting these parents, a newly-arriving client is able to start downloading the substreams into different buffers. The software on the client side includes a thread that reads from these buffers in a round robin fashion and writes to a local port using an HTTP server. Any media players with the ability to play streams over HTTP (*e.g.*, `mplayer`) can play out the stream from this local port. Similar to the software on the provider side, the software at the client also starts a listener process, which instantiates threads in response to join request from children. When such a request is received for a specific substream, the software sends data from the buffer associated with that substream. The angels' software is similar to the software at the clients, except that angels need only subscribe to one substream and then listen and serve incoming client requests for that one substream.

Figure 9 illustrates an example of the interaction between a newly arriving client, client *A*, the registrar and other clients chosen by the registrar as parent to client *A* in two trees. Client *A* joins the system by sending a "Join" message (message #1) to the registrar, the "Join" message contains the ID of the requested stream and the upstream capacity it is willing to contribute. The registrar replies with a "Welcome" message (message #2) informing the client with the stream playout rate, the number of trees (substreams), the chunk size, and the identity of the chunk it should download first. The registrar also sends to the client the IP/port# of each parent that the client should contact for each substream (Messages #3 and #5). When contacting these parents (messages #4, #6), the client specifies the tree ID and the chunk it is expecting to start downloading from. Messages #7 to #13 represent data messages from the parents to client *A*. The data messages has the chunkID and the associated streaming data. In this example, the streaming continues smoothly (message #13), after that the connection is lost or degraded (message #14 is lost). The client realizes that and contacts the registrar before it depletes its buffer. It sends message #15 to the registrar informing it that it was disconnected from its parent in the first tree, client *B*. The registrar replies to client *A* with a new parent to connect to, client *D* (Message #16). The client sends message #17 to the new par-

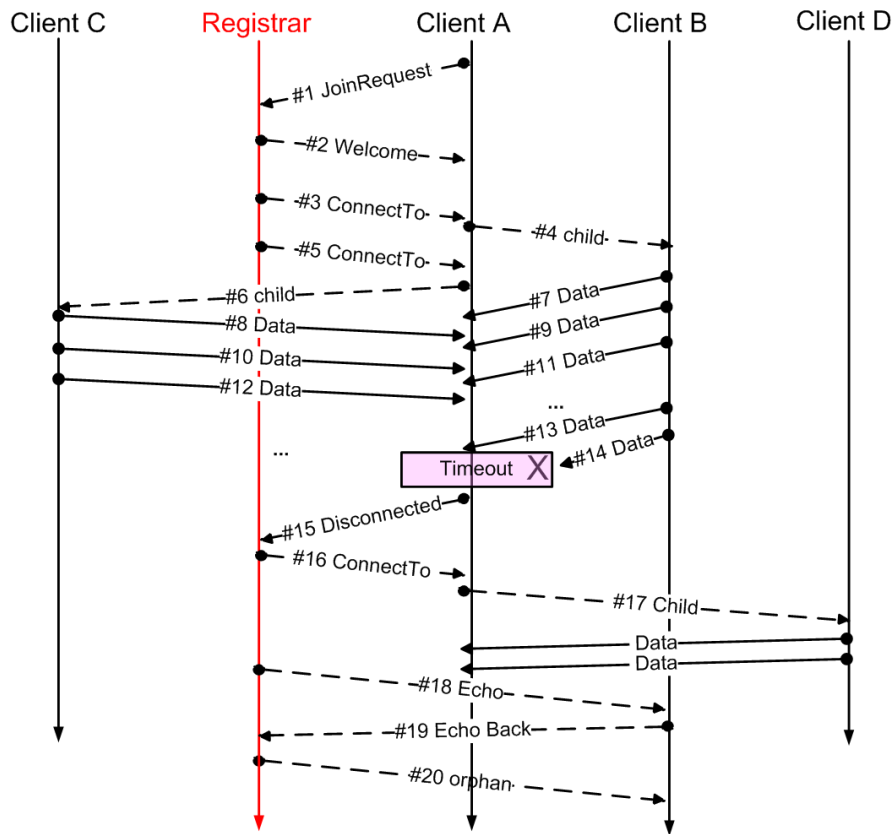


Figure 9: An interaction diagram showing the exchange of messages between a new client, the registrar and the parents.

ent, client *D*, requesting chunks starting at where it stopped, the new parent starts streaming this substream. In the meanwhile, the registrar probes old parent, client *B*, to check if it is still alive or not (Message #18). In this case it receives an "EchoBack" message (#19), and the registrar sends the old parent message #20 informing it to disconnect that child. If the registrar had not received an "EchoBack" message, it would have assumed the client is disconnected and would have sent a proactive message to all its children in all trees to connect to a different parent. For security reasons, the registrar sends messages to the parents informing them which clients to accept as children. The client can ignore any download request from any unauthorized client. We omitted those messages from this example for simplicity.

We implemented a prototype of our *AngelCast* protocol in python.<sup>4</sup> Our prototype includes the code for the registrar, provider, clients and angels. Our prototype does not include the profiler, thus clients report how much upstream capacity they are willing to contribute to each swarm.

## 5. Experimental Evaluation

To evaluate the performance of our *AngelCast* prototype, we deployed it in the widely used research platforms: Emulab [8] and PlanetLab [27]. On the one hand, the Emulab experiments give us accurate insights by isolating our protocol from other experiments, which is particularly useful to analyse the effects of churn, and also making it possible for our results to be repeatable. On the other hand, the PlanetLab experiments are meant to validate that *AngelCast* performs well “in the wild” on the Internet.

Our main motivation in performing these experiments is four-fold: (1) establish confidence in our implementation by comparing its performance to that of widely used streaming solutions, (2) establish the effectiveness of deploying angels from the cloud for the purpose of guaranteeing the desired streaming rates, (3) measure the performance of our system under churn, and (4) study the effectiveness of setting Angelcast system’s parameters. We deploy the registrar on a machine of its own. The angels are deployed on Emulab/PlanetLab machines instead of the cloud. All results are reported with 95% confidence interval.

The first set of experiments aims at validating our *AngelCast* prototype by comparing its performance to that of SopCast [30]. SopCast is a popular P2P streaming client used widely on the Internet. We ran SopCast and *AngelCast* protocols on the same set of machines at the same time to neutralize unpredictabilities related to host/network processes (*e.g.*, cross-traffic).

We performed experiments to compare the frame drop-ratio of *AngelCast* vs Sopcast for streams of varying rates (280 Kbps to 1.4 Mbps) on Planetlab. We use traffic shapers to limit the upstream capacity of the nodes which had significantly higher upstream bandwidth than the average household. We limit the upstream capacity of the provider to twice the stream rate,  $2 * r$ , and clients upstream capacity to  $(4/3) * r$ . This assignment guarantees that there is enough upstream capacity for all clients, thus, there is no need for

---

<sup>4</sup>Available at:<http://csr.bu.edu/angelcast/AngelCast/>

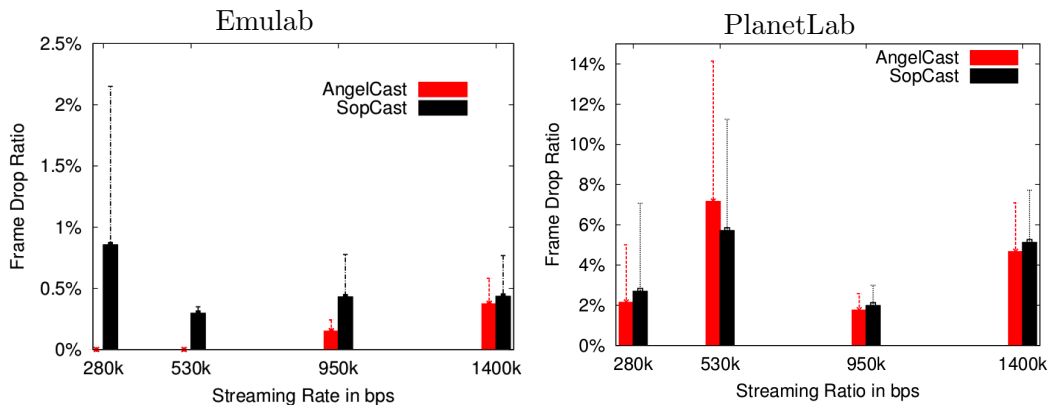


Figure 10: The frame drop ratios of *AngelCast* vs *SopCast*.

angels. The metric which we utilize throughout the experiments is frame drop ratio which for the correctness includes not only the frames that did not arrive to the clients, but also the frames that did not arrive on time. In this baseline experiment, the stream is split into ten substreams each is disseminated through a trinary tree. The start-up buffer between the time a client requests to join a stream and the start of the playout is four seconds. The result in Figure 10a shows that the frame drop-ratios of *AngelCast* and *SopCast* are comparable when the upstream capacity is plentiful.

Figure 10b shows the result of the same experiment on PlanetLab. The frame drop-ratio is significantly higher than in Emulab but the performance of *AngelCast* and *SopCast* is still comparable. This is expected, since the benefits gained from *AngelCast* will only be observed when there is a significant gap between the clients’ aggregate upload and download capacity. Because Emulab allow us to perform repeatable experiments and to isolate the performance from other experiments running on the same machine, we decided to run the rest of the experiments on Emulab.

The second set of experiments aims to characterize the effectiveness of angels. Our system deploys angels when a tree has no vacantSpots. We secured 125 clients for downloading a live stream at  $r=950\text{Kbps}$  playout rate. We vary the client upstream capacity between 60% to 100% of the stream rate,  $r$ . *AngelCast* splits the stream into ten trees each with a fan-out of three. The provider’s upstream capacity is double the stream rate, ensuring that the provider is not the bottleneck. An angel upstream capacity is 1.5

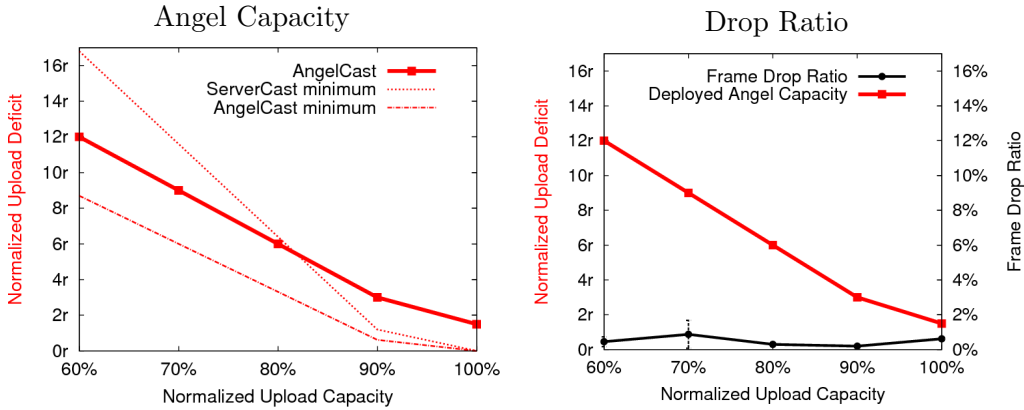


Figure 11: Minimal amount of angel capacity is sufficient to achieve good stream rates.

times the stream rate, ensuring that it is larger than the stream rate. but also that it does not have too many children in one tree (maximum=15).

On the x-axis of Figures 11a and 11b, we vary the client upstream capacity, shown as the ratio between the client upstream capacity and the stream rate. On the y-axis of Figure 11a we plot the capacity of angels deployed by *AngelCast* against the theoretical bound for the minimum angel capacity (*AngelCast* theoretical). Also, we compare it against the minimum server capacity when the server downloads the whole live stream (ServerCast). The results confirm that *AngelCast* utilizes near minimal capacity, and that it achieves significant savings when compared to ServerCast. On the left-hand-side of Figure 11b (in red), we plot the angel capacity being used and on the right-hand-side scale (in black), we plot the associated frame drop ratio. This experiment verifies that our system achieves reliable streaming utilizing near minimal resources.

The third set of experiments aims to demonstrate the performance of *AngelCast* under churn. In live streaming, churn is due to client arrivals and departures. This is different from how churn is typically modeled in VoD where playback functionality, such as pause, seek and fast-forward must be considered as well [10]. Therefore in this experiment, we observe a live stream over a period of 210 seconds, whereby clients join the stream after an exponentially distributed waiting period with a mean of ten seconds. Clients watch the stream for an exponential amount of time of mean 50, 100, 150, 200 or 250 seconds then leave.

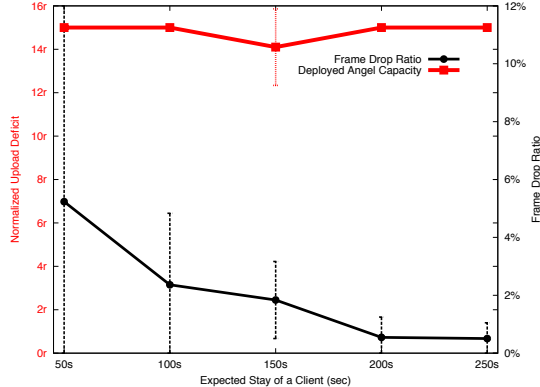


Figure 12: The performance of *AngelCast* under churn.

We set the upstream capacity of the provider to be twice the playout rate; we set the upstream capacity of angels to be 1.5 times the playout rate; and we set the clients' upstream capacity to be 0.7 times the playout rate. The stream is divided over ten trinary trees and the stream start-up buffer is 4 seconds. The first set of results are in Figure 12, on the x-axis we vary the expected duration of the client's stay. On the right y-axis we show the frame drop-ratio. This experiment shows that, as expected, higher churn results in poor performance, (*e.g.*, when clients stay less than a minute on average, the frame drop ratio is as bad as 5% but as clients stay longer, the frame drop ratio drops to 0.5%). On the left y-axis, we plot the aggregate capacity of the deployed angels. When there is no churn, six angels are needed to fill the capacity gap. In the presence of churn, the number of deployed angels is almost fixed to ten, the number of trees. The reason for that is the lack of vacantSpots in each tree at different points during the experiment, due to churn.

In the aforementioned experiment, clients leave the system and do not rejoin. Stutzbach and Rejaie observed that some clients who leave the stream, rejoin it [32]. Thus, in Figure 13 we show results for the same experiment where a departing client rejoins the stream after an average of 10 seconds waiting time. The results show similar performance to the ones shown in Figure 12 – mainly a frame drop ratio as bad as 5% under heavy churn,

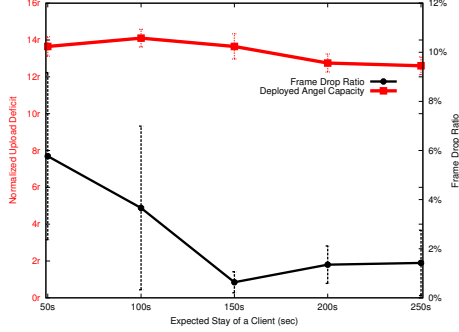


Figure 13: The performance of *AngelCast* under churn where clients rejoin the stream after they leave.

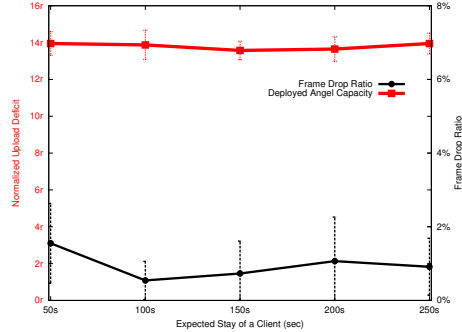


Figure 14: The frame drop ratio for non-churning clients against varying churning rate for other clients.

which drops significantly as churn rate decreases.

Our next experiment studies the effect of churning clients on the performance of non-churning clients (*i.e.*, clients who stay for the whole duration of the experiment). Given the similarity in the performance of churn models (cf. Figures 12 and 13), we select the client departure model to be the one where clients leave and do not return and measure the frame drop ratio of non-churning clients. Figure 14 shows the results obtained while varying the expected duration of a churning client’s stay. The left y-axis, shows the aggregate capacity of the deployed angels, while the right y-axis shows the frames drop-ratio for *non-churning* clients. The results show that the performance of non-churning clients is independent of the churn level of other clients. It also illustrates that not only our system’s tolerance for churn, but also highlight that the resulting modest degradation is confined largely to churning clients.

The final set of experiments aims at studying the effect of setting *AngelCast* parameters on the performance of clients. We evaluate two system level parameters: start-up buffer size and number of substreams (trees). Figure 15 shows the result of an experiment studying the effect of the start-up buffer on the performance of clients. The x-axis denotes the start-up delay in seconds and the y-axis denotes the frame drop ratio. We plot the results of a family of varying churn levels, where clients stay for a duration of 50, 100, 150, 200 or 250 seconds (*i.e.* no churn). The results highlight that three or four seconds are enough to achieve smooth streaming. Longer buffers will

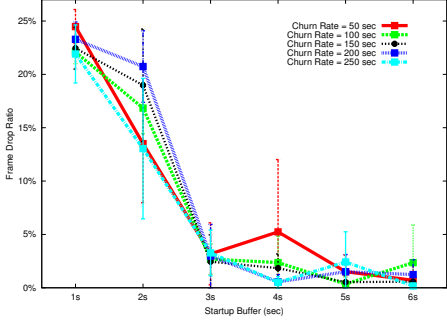


Figure 15: The frame drop ratio of clients vs. start-up buffer size. Different curves denote different churning levels.

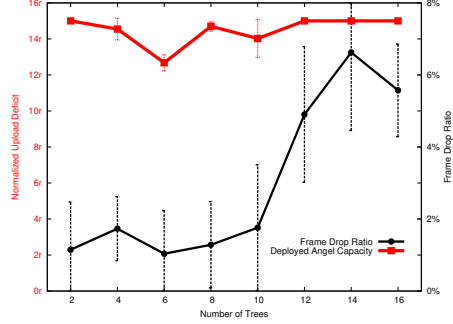


Figure 16: The frame drop ratio of clients against an increasing number of substreams (trees).

result in stale content without achieving meaningful performance gains.

Figure 16 presents results highlighting the effect of varying number of substreams (trees) on the performance clients. The x-axis denotes the number of such trees, the right y-axis denotes the frame drop ratio, and the left y-axis denotes the normalized angel capacity utilized. We set the number of available angels to ten. When the number of trees is less than ten, the frame drop ratio is low. In fact, the optimal value for this specific configuration is six trees, as the frame drop ratio is lowest and we use the least number of angels. When the number of trees is greater than ten, some trees will not be assigned angels. Thus in the event of churn there will be significant frame loss in the angel-less trees despite having enough overall angel capacity to deliver the content. We conclude that we should limit the number of trees to a handful to avoid wasting angel capacity or having high frame drop ratio in the angel-less trees.

In our experiments, the *AngelCast* system was able to handle a cluster of Emulab nodes. In a typical P2P stream setting, there may be more nodes under management. We note that the architecture of our *AngelCast* system is scalable by design as an embarrassingly parallel system. In particular, Angels can be allocated from the cloud which can scale based on demand. The registrar and accounting services (cf. Figure 6) can be implemented as services on a Mesos/Marathon cluster while node information can be kept in a DHT or NoSQL database to ensure scalability.

## 6. Related Work

This work builds on a rich body of work that approach the problem of content delivery in general and live streaming in particular from a number of perspectives:

**Pull-based Mesh Protocols:** There is a large number of papers/systems that utilize pull-based mesh streaming, such as SopCast, PPLive, UUsee, Joost, and CoolStreaming. Our push-based approach differs from these in that it choreographs the connectivity among nodes to guarantee the quality of streaming for *every* client. We choose to compare *AngelCast* against SopCast [30] as it is extensively used, allows users to stream their own channels and works with `mplayer` over Linux. An example of such pull-mesh protocols is CoolStreaming [38], the streaming version of Bittorrent. The difference is that the deadline of a chunk playtime is a factor in the piece selection algorithm. Feng et al. [9] illustrated the inherent shortcomings of pull-based mesh networks as well as providing a glossary of such protocols.

**Peer-Assisted Content Distribution:** The research community is aware of the promise of P2P in alleviating the load of content distribution on servers. Nonetheless, it is also aware of its limitations, especially in providing sufficient upstream capacity. For file sharing, Sweha et al. [33], introduced the idea of angels to minimize the bulk download time for a fixed group of clients. Wang et al. [36] proposed adding powerful peers to BitTorrent swarms to accelerate the download rate. This is not optimal, because these “helpers” will unnecessarily consume the scarce upstream capacity in the swarm. Montresor and Abeni [18] employs a single passive helper and enforces strict limits on the number of (costly) interactions with it that originate from peers. In Antfarm [26], the authors propose a protocol for seeders to measure the vital signs for multiple swarms and allocate more seeder bandwidth to struggling swarms. Jin et al. [12] introduced edge caching to help original servers stream to clients at the prescribed bit-rate. The cache contains the objects where the ratio between the client request rate and the deficit between download rate and playback rate is maximal. This work is different as it requires in-network caching, but it highlights the need for infrastructure help to ensure smooth streaming.

**Modeling P2P Performance:** Qiu and Srikant’s seminal work [28] is the first to characterize the average download time in a swarm. Das et al. [6] extended Qiu’s model [28] of swarm download rate to incorporate seeders. The result points out that the average download time is inversely proportional

to the seeders' aggregate upload rate. However, the increase in the number of peers requires a linear increase in the number of seeders to maintain the same average download time. Although Qiu's model is fundamentally different, these results are in line with our findings. Parvez et al. [25] extended Qiu and Srikant's model [28] to the case of stored VoD, studying the need for sequential progress instead of random chunk download. This is inline with our design concept of choreographing the connectivity of the swarm instead of relying on random chunk download.

Kumar et al. [15] used the uplink sharing model [19] to find the bound on the highest streaming rate a swarm can handle given the upstream/downstream capacities of the clients. They proposed a fluid construction that achieve their bound. We extend their model to compute the required angel capacity to achieve a desired streaming rate. Likewise, we built an optimal fluid construction that achieves the bound, incorporating angels. They highlighted the gap between the upstream capacity and download rate, stating that most channels in PPLive are running at rate 2-4 times the upstream capacity of many residential broadband peers. Angels are the answer to this problem. Liu et al. [17] extended Kumar's model in the case of bounded node degree as well. They proposed creating spanning trees with varying streaming capacities. Given their construction, they provided bounds on the depth of the tree, the maximal upload rate and the minimum server capacity. Their constructions assumes the provider can connect to all clients, that the number of spanning trees can reach the number of clients and the depth of some trees can be linear in the size of the swarm. Our *AngelCast* technique avoids these three problems.

**Multicast Multi-Tree Construction:** Many papers leverage the idea of constructing multiple multicast trees (a forest) for file distribution as well as for streaming [35]. SplitStream [4] constructs a forest of multicast trees, one for each stripe, all with the same rate. This approach is different from ours in that the client can choose the number of stripes it wants to subscribe to, thus receiving a subset of the broadcasted data. In contrast to tree-based multicast, the load on the nodes is balanced as each node is an internal node in one tree but a leaf node on the others. When the internal nodes of one tree cannot adopt any more children, the child must search for some node in the excess capacity tree to download this strip. This leads to inefficiencies as a node could perform a distributed linear search until it finds a suitable parent. More importantly a client with significantly large capacity can adopt many children each in a different tree resulting in degenerate trees and more

than logarithmic depth of trees. Our technique makes sure that any parent has at least two children except for the nodes in the second to last level, insuring a logarithmic depth of all trees.

P2PCast [21] is a modification of SplitStream where all nodes subscribe to all trees to download the same content. They require from each client to participate an upstream capacity equal to the download rate. This could be unrealistic given the diversity of user’s upstream capacities and the shielding of some clients behind NATs. CoopNet[24, 23] creates multiple trees, each streaming a substream of an MDC encoded video. Their contribution is in providing a mechanism to cope with the fluctuation in the available bandwidth.

**Coding for Adaptive Streaming:** Multiple Description Coding (MDC)[5] and Scalable Video Coding (SVC) [1] were introduced to enable clients to download the same video on different rates/qualities. MDC tends to be more theoretical, where any number of descriptions are enough to decode the movie at a rate proportional to the number of received descriptions. SVC codes the video in layers, the base layer is essential for decoding while the subsequent layers are increasingly less important. Such techniques are increasingly deployed to overcome the uncertainty of the available bandwidth and its fluctuation, in particular Dynamic Adaptive Streaming over HTTP (DASH) [31]. We consider these techniques complementary to our angels approach. Providers who prefer better than *best effort* delivery to their clients can deploy angels to offer better download rate. MDC and SVC can be deployed in conjunction with angels in this case, where clients can attempt subscribing to  $m$  trees downloading  $m$  descriptions/layers. If a client is not able to download all the layers, it is still able to decode the video with lower quality. Our current prototype implementation works over HTTP, we are planning on extending our protocol to conform more closely to DASH.

**Hybrid Cloud-P2P Systems:** Existing work in P2P with cloud assistance deal mainly with large number of clients and focus on improving the aggregate system performance [13, 39, 37, 14, 34, 7, 16]. Jin and Kwok [13] present a cloud assisted system architecture for P2P media streaming among mobile peers to minimize energy consumption. Zhao et al. [39] use a loss network model to tradeoff between inter-ISP traffic and cloud bandwidth consumption and propose a hybrid P2P-cloud CDN system. Wang et al. [34] presented a generic framework called CALMS for the migration of live streaming services to the cloud, which adaptively leases and adjusts cloud resources to meet dynamic user demands. CloudStream [11] considered re-

alttime transcoding of videos in different qualities over cloud and delivered video streams via a cloud-based SVC proxy. The authors in [7] propose an architecture that adapts dynamically the playback rate to guarantee that peers receive the stream even in cases where the total upload bandwidth changes very abruptly. Our work focuses on using cloud resources as *Angels* to supplement the persistent gap that exists between the average downlink and the average uplink capacity of peers.

## 7. Conclusion

In this paper we highlighted the potential of peer-assisted content distribution for affordable high quality live streaming. As the deficit between clients' uplink and downlink capacities limits the use of pure P2P architecture in such a setting, we introduced the notion of angels – servers who do not have a feed of the live stream and are not interested in downloading it in full. We computed the minimum amount of angel capacity needed in a swarm to achieve a certain bit-rate to all clients and provided a fluid model construction that achieves that bound. We introduced practical techniques that handle limited node degree constraints and churn. We built *AngelCast*, a cloud-based service that assists content providers in delivering quality streams to their customers, while allowing the content providers to leverage the customers' resources to the fullest. We deployed *AngelCast* unto two research platforms: Planetlab and Emulab. We showed that the performance of *AngelCast* is comparable to that of SopCast, a widely used streaming protocol, and that it is capable of supplementing the bandwidth deficit with near minimal capacity, while being able to handle churn. We are currently developing a version of *AngelCast* for wide-spread deployment. We are planning on collecting measurements of its performance in delivering real-world live streams. Our future work will explore ways in which under-utilized angels are managed. This includes the possibility of using angels across multiple swarms. Security is another focus of our future work, especially securing the registrar control messages.

## References

- [1] Abboud, O., Zinner, T., Pussep, K., Sabea, S. A., Steinmetz, R., 2011. On the impact of quality adaptation in SVC-based P2P video-on-demand systems. ACM 2nd MMSys '11.

- [2] Akamai, 2016. Akamai Netsession. <http://www.akamai.com/client/>.
- [3] Bittorrent, 2016. Bittorrent DNA. Bittorrent DNA, <http://www.bittorrent.com/dna>.
- [4] Castro, M., Druschel, P., Kermarrec, A., Nandi, A., Rowstron, A., Singh, A., 2003. Splitstream: high-bandwidth multicast in cooperative environments. In: SOSP.
- [5] Chou, P., Wang, H., Padmanabhan, V., 2003. Layered multiple description coding. In: Proc. Packet Video Workshop. Citeseer.
- [6] Das, S., Tewari, S., Kleinrock, L., 2006. The Case for Servers in a Peer-to-Peer World. Communications, IEEE International Conference on.
- [7] Efthymiopoulou, M., Efthymiopoulos, N., Christakidis, A., Athanasopoulos, N., Denazis, S., Koufopavlou, O., 2015. Scalable playback rate control in p2p live streaming systems. Peer-to-Peer Networking and Applications, 1–15.
- [8] Emulab, 2016. <https://www.emulab.net/>.
- [9] Feng, C., Li, B., Li, B., 2009. Understanding the performance gap between pull-based mesh streaming protocols and fundamental limits. In: INFOCOM 2009, IEEE. pp. 891–899.
- [10] Guo, Y., Yu, S., Liu, H., Mathur, S., Ramaswamy, K., 2008. Supporting vcr operation in a mesh-based p2p vod system. In: CCNC 2008. 5th IEEE. pp. 452–457.
- [11] Huang, Z., Mei, C., Li, L., Woo, T., 2011. Cloudstream: Delivering high-quality streaming videos through a cloud-based svc proxy. In: INFOCOM, 2011 Proceedings IEEE. IEEE.
- [12] Jin, S., Bestavros, A., Iyengar, A., 2002. Accelerating internet streaming media delivery using Network-Aware partial caching. ICDCS'02, 153+.
- [13] Jin, X., Kwok, Y.-K., 2010. Cloud assisted p2p media streaming for bandwidth constrained mobile subscribers. In: Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on. IEEE.

- [14] Kavalionak, H., Carlini, E., Ricci, L., Montresor, A., Coppola, M., 2015. Integrating peer-to-peer and cloud computing for massively multiuser online games. *Peer-to-Peer Networking and Applications*.
- [15] Kumar, R., Liu, Y., Ross, K., 2007. Stochastic fluid theory for P2P streaming systems. In: *INFOCOM 2007. 26th IEEE International Conference on Computer Communications*.
- [16] Liu, J., Ahmad, S., Buyukkaya, E., Hamzaoui, R., Simon, G., 2015. Resource allocation in underprovisioned multioverlay peer-to-peer live video sharing services. *Peer-to-peer networking and applications*.
- [17] Liu, S., Zhang-Shen, R., Jiang, W., Rexford, J., Chiang, M., 2008. Performance bounds for peer-assisted live streaming. In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 36.
- [18] Montresor, A., Abeni, L., 2011. Cloudy weather for p2p, with a chance of gossip. In: *IEEE Conference on Peer-to-Peer Computing (P2P)*.
- [19] Mundinger, J., Weber, R., Weiss, G., 2008. Optimal scheduling of peer-to-peer file dissemination. *Journal of Scheduling* 11 (2), 105–120.
- [20] Netflix, January 2016. Usa isp speed index. <https://media.netflix.com/en/company-blog/>.
- [21] Nicolosi, A., Annapureddy, S., 2003. P2pcast: A peer-to-peer multicast scheme for streaming data. In: *1st IRIS Student Workshop ISW*.
- [22] Octoshape, 2016. <https://www.octoshape.com/>.
- [23] Padmanabhan, V., Wang, H., Chou, P., 2005. Supporting heterogeneity and congestion control in peer-to-peer multicast streaming. *Peer-to-Peer Systems III*, 54–63.
- [24] Padmanabhan, V. N., Wang, H. J., Chou, P. A., 2003. Resilient peer-to-peer streaming. In: *11th IEEE ICNP'03*.
- [25] Parvez, N., Williamson, C., Mahanti, A., Carlsson, N., 2008. Analysis of bittorrent-like protocols for on-demand stored media streaming. *SIGMETRICS'08*.

- [26] Peterson, R. S., Sizer, E. G., 2009. Antfarm: efficient content distribution with managed swarms. In: NSDI. USENIX Association.
- [27] PlanetLab, 2016. <https://www.planet-lab.org/>.
- [28] Qiu, D., Srikant, R., 2004. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. SIGCOMM Comput. Commun. Rev. 34.
- [29] Salehi, J., Zhang, Z., Kurose, J., Towsley, D., 1996. Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing. ACM SIGMETRICS Performance Evaluation Review 24 (1), 222–231.
- [30] Sopcast, 2016. Free P2P Broadcasting, <http://www.sopcast.org/>.
- [31] Stockhammer, T., 2011. Dynamic adaptive streaming over http-: standards and design principles. In: ACM 2nd MMSys'2011.
- [32] Stutzbach, D., Rejaie, R., 2006. Understanding churn in peer-to-peer networks. In: Proceedings of IMC.
- [33] Sweha, R., Ishakian, V., Bestavros, A., 2011. Angels In The Cloud: A Peer-Assisted Bulk-Synchronous Content Distribution Service. In: The IEEE Conference on Cloud Computing.
- [34] Wang, F., Liu, J., Chen, M., 2012. Calms: Cloud-assisted live media streaming for globalized demands with time/region diversities. In: INFOCOM, 2012 Proceedings IEEE. IEEE.
- [35] Wang, F., Xiong, Y., Liu, J., 2007. mtreebone: A hybrid tree/mesh overlay for application-layer live video multicast. In: ICDCS. IEEE.
- [36] Wang, J., Yeo, C., Prabhakaran, V., Ramch, K., 2007. On the role of helpers in peer-to-peer file download systems: Design, analysis and simulation. In: In IPTPS'07.
- [37] Wu, Y., Wu, C., Li, B., Qiu, X., Lau, F. C., 2011. Cloudmedia: When cloud on demand meets video on demand. In: ICDCS. IEEE.

- [38] Zhang, X., Liu, J., Li, B., Yum, T., 2005. Coolstreaming/donet: A data-driven overlay network for efficient live media streaming. In: proceedings of IEEE Infocom. Vol. 3. pp. 13–17.
- [39] Zhao, J., Wu, C., Lin, X., 2014. Locality-aware streaming in hybrid p2p-cloud cdn systems. Peer-to-Peer Networking and Applications.