ShadowStream: Performance Evaluation as a Capability in Production Internet Live Streaming Networks

Chen Tian* Richard Alimi† Yang Richard Yang*× David Zhang§
*Yale University †Google §PPLive ×SplendorStream
{tian.chen, yang.r.yang}@yale.edu ralimi@google.com davidzhang@pplive.com

ABSTRACT

As live streaming networks grow in scale and complexity, they are becoming increasingly difficult to evaluate. Existing evaluation methods including lab/testbed testing, simulation, and theoretical modeling, lack either scale or realism. The industrial practice of gradually-rolling-out in a testing channel is lacking in controllability and protection when experimental algorithms fail, due to its passive approach. In this paper, we design a novel system called ShadowStream that introduces evaluation as a built-in capability in production Internet live streaming networks. ShadowStream introduces a simple, novel, transparent embedding of experimental live streaming algorithms to achieve safe evaluations of the algorithms during large-scale, real production live streaming, despite the possibility of large performance failures of the tested algorithms. ShadowStream also introduces transparent, scalable, distributed experiment orchestration to resolve the mismatch between desired viewer behaviors and actual production viewer behaviors, achieving experimental scenario controllability. We implement ShadowStream based on a major Internet live streaming network, build additional evaluation tools such as deterministic replay, and demonstrate the benefits of ShadowStream through extensive evaluations.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement techniques.
General Terms: Measurement, Experimentation, Design.
Keywords: Performance Evaluation, Live Testing, Streaming.

1. INTRODUCTION

Motivation: Live streaming (*e.g.*, [11, 27, 45]) is a major Internet application that has been used to carry both daily and major events such as the Obama inauguration address, the 2010 Winter Olympics, and the 2010 World Cup. As live streaming continues to permeate into our daily lives [1], it is becoming increasingly important that Internet live streaming networks provide reliable performance, despite such networks becoming more complex, to operate in the increasingly more complex Internet.

A key capability to ensure that live streaming networks provide reliable performance is to subject them to large-scale, realistic performance evaluations. In our recent survey of the developers of a number of Internet-scale application sites, the ability to conduct large-scale, realistic performance evaluations is among the most desired, but the most difficult to achieve abilities to improve site reliability. In the context of live streaming, without such an ability, developers have to resort to theoretical modeling (*e.g.*, [8, 25, 46]), simulation (*e.g.*, [29, 39]), or lab/testbed testing (*e.g.*, [6, 10, 22, 31, 32]). But all of these existing methods are seriously limited either in scale or in realism. For example, no existing testbed can scale easily to tens of thousands of clients but many live streaming events have reached or even surpassed this scale. Further, it is becoming increasingly difficult for modeling, simulation, or testbed to capture the increasingly complex Internet, which includes diverse ISP network management practices and heterogeneous network features such as PowerBoost [2, 36] in cable networks, large hidden buffers in access networks [15, 23, 36], and shared bottlenecks at ISP peering or enterprise ingress/egress links [24].

A practice used by the live streaming industry is to use a testing channel to roll out new versions. However, the passive approach of this practice leads to serious limitations on scale and experimental feasibility. First, due to concerns on the failures of experimental algorithms, existing usage of testing channels is conservative, limiting tests to a small number of real users. Second, using real viewers as they naturally arrive and depart, may not provide the target operating conditions that stress the experimental system. A developer may always want to evaluate whether a modification will change the effectiveness of the live streaming system to handle a flash-crowd with 100,000 viewers. Although the testing channel may currently have more than 100,000 real viewers, the natural arrivals of the viewers do not conform to a flash-crowd arrival pattern.

Live streaming systems updated without going through realistic, large-scale evaluations may operate at sub-optimal states, and often do not achieve performance expectations formed at small-scale lab settings. For example, PPLive, a major live streaming distribution network, encountered surprising performance failures when it moved from an initial deployment in the university-based CERNET (China Education and Research Network) to the general Internet with many viewers in ADSL networks. A major category of viewer complaints in the viewer forum of PPLive [33] is poor performance after network updates.

The motivation of designing ShadowStream starts with a relatively extreme recognition that a production live streaming network can be a rarely available foundation for designing a testing system with both *scale* and *realism*, by taking advantage of its large number of real viewers distributed across the real Internet. Building on this foundation to introduce protection and controllability, we develop a novel Internet live streaming system that integrates performance evaluation as an intrinsic capability, a new capability that complements analysis, simulation, and lab testing. We refer to testing during production streaming as *live testing*.

Challenges: A live testing platform poses two key challenges that are not present in traditional testing platforms such as VINI [7], PlanetLab [12], or Emulab [41]. The objective of ShadowStream is to address both challenges.

The first is *protection*. Since real viewers are used, live testing needs to protect the real viewers' quality of experience from the performance failures of experimental systems. Hence, live testing needs to manage the challenge of letting performance failures of an experimental system happen so that poor performing algorithms can be identified, and at the same time masking these failures from real viewers. As a contrast, a traditional testing platform uses ded-

SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.

icated clients in a non-production environment, and hence performance failures of an experimental system will not do harm.

The second is *orchestration*. In particular, live testing needs to orchestrate desired experimental scenarios (*e.g.*, flash-crowd) from production viewers, without disturbing their quality of experiences. In a traditional testing platform, clients are dedicated and fully controllable artificial viewers. In live testing, a real viewer expects that the live streaming service starts promptly, as soon as the viewer arrives; real viewers do not expect disruptions in the middle of the streaming to ask them to join experiments; as soon as a real viewer closes the live streaming program, the service should stop, not continuing as a hidden background process to finish the experiment.

Our approach: ShadowStream applies a general *Experiment→ Validation→Repair* scheme to achieve protection and transparent orchestration; see Section 8 on discussions for generalization. Specifically, ShadowStream introduces, at each client, a simple *streaming hypervisor*, which manages multiple concurrent *virtual streaming machines*, one of which is an experimental streaming machine containing the experimental algorithms. The ShadowStream hypervisor first assigns each task (*i.e.*, downloading a video piece) to the experiment, as if the experimental system were running alone, and records any failures (*i.e.*, if a piece does not arrive before the target deadline of the experimental system), to obtain accurate experimental results. However, repair steps in to fix experimental failures before they become visible to real viewers. The hypervisor also implements *pre-reservation* of task space so that an experimental system can always be started without the interference of the production system (*i.e.*, inserting an experiment while the real viewer has already started with the production system). To make repair and pre-reservation feasible, in the context of hybrid live streaming systems, ShadowStream introduces a small additional lag on the playpoints of real viewers.

ShadowStream extends the basic scheme with novel algorithms to achieve scalability, for both protection and orchestration. In particular, it introduces a novel *production-CDN-experiment (PCE)* streaming machine layout, to achieve highly scalable protection; it also introduces novel, local orchestration algorithms to achieve highly scalable orchestration.

We implement ShadowStream based on one of the largest live streaming networks. Our implementation shows that ShadowStream is easy to implement, with code base around 8,000 lines. Based on real system traces, we demonstrate that ShadowStream can allow scalable testing reaching 100,000 clients while no existing testbed can easily reach even thousands.

2. MOTIVATION

2.1 Live Streaming Grows in Complexity

Many modern live streaming networks have become complex hybrid systems that use both peer-to-peer (P2P) networks and content delivery networks (CDN) [30]. Adobe Flash is one platform that has implemented support for a hybrid architecture in recent versions. In this paper, we target this general, widely used streaming architecture.

The basic design of a hybrid live streaming network is typically BitTorrent-like. Guided by a tracker, peers viewing the same live event connect with some neighbors to form an overlay network to relay video data through the overlay topology. A source encodes small clips of video data called *pieces*, which are made available to CDNs. Some peers in the overlay network download pieces from CDN networks. A peer may directly push new pieces to some neighbors and/or exchange piece bitmaps with its neighbors so that the neighbors can request pieces from it.

To handle practical challenges and improve performance, the preceding basic design is typically substantially enhanced. Hence, a production live streaming system often includes a large number of modules: (i) P2P topology management, (ii) CDN management, (iii) buffer and playpoint management, (iv) rate allocation, (v) download/upload scheduling, (vi) viewer-interfaces, (vii) shared bottleneck management (*e.g.*, [20]), (viii) flash-crowd admission control, and (ix) network-friendliness (*e.g.*, [5, 44]).

The performance of a network with a large number of peers each running a software system containing many modules can become difficult to evaluate. A main metric on the performance of live streaming is *piece missing ratio*, which is the fraction of pieces that are not received by their playback deadlines. A major factor affecting piece missing ratio is channel *supply ratio*, which is the ratio between total bandwidth capacity (CDN and P2P together) and total streaming bandwidth demand of all participating peers. Next, we use a few concrete examples to demonstrate the importance of large-scale realistic experiments.

2.2 Misleading Results at Small-Scale

Testbed experiments have difficulty replicating the scale of actual deployment, which can reach hundreds of thousands of viewers. But results obtained from small-scale networks can be quite different from those obtained from large scale networks. As a concrete example, we run the same production version (default version) of a major live streaming network at two scales in Emulab: 60 clients (smaller scale) and 600 clients (larger scale). In both cases, clients join sequentially with a 5-second interval; the streaming rate is 400 Kbps; each client's upload capacity is 500 Kbps, and a CDN server provides an upload capacity of 10,000 Kbps. Thus, the supply ratio of the smaller network is around 1.67 (= (60 * 500 + 10,000)/(60 * 400)), while for the larger network it is around 1.29. Since the smaller network has a larger supply ratio, one might think that it would perform better than the larger network.

	Small-Scale	Large-Scale
Production Default	3.5%	0.7%

Table 1: Piece missing ratio at different scales (default).

Table 1 shows the performance of the two networks measured by piece missing ratio. We observe a large performance difference: the piece missing ratio of the small network is 5 times that of the large network. A detailed trace analysis shows that content bottleneck (*i.e.*, diversity of data among neighbors) is a key reason for the poor performance of the small network. The average buffer map difference among peers is not sufficient to fully utilize available bandwidth. On the other hand, in the larger network, content bottleneck may not be an issue; the peer upload bandwidth together with the bandwidth scheduling algorithm dominate the performance. This observation is consistent with a recent measurement [42].

	Small-Scale	Large-Scale
With Connection Limit	3.7%	64.8%

Table 2: Piece missing ratio at different scales (variant).

We run a variant of the default version with a connection limit (*i.e.*, a client declines all subsequent connection requests when the connection limit is reached). Table 2 shows a surprising and interesting result: after imposing the limit, we again observe a large performance difference, but opposite from the preceding result. Specifically, the performance degrades significantly in the larger scale experiment, with piece missing ratio of the larger scale experiment being 17.5 times that of the smaller scale! Our detailed evaluation shows that with a connection limit, the CDN server's neighbor connections are exhausted by those clients that join earlier and the clients that join later cannot connect to the CDN server and experience starvation. Both experiments illustrate that observations from a smaller scale may not directly translate to those at a larger scale.

2.3 Misleading Results due to Missing Realistic Features

It is extremely challenging to capture all aspects of real networks in a simulator, emulator, or testbed. For example, end hosts may differ in type of their network connectivity, amount of system resources, and network protocol stack implementations; routers may differ in their queue management policies, scheduling algorithms, and buffer sizes; background Internet traffic may change dynamically and is hard to replicate. Failures to capture any of these effects may result in misleading results. As a motivating example, below we show that simply changing the access link types can lead to different performance results.

Specifically, typical academic testbeds (*e.g.*, PlanetLab, Emulab) are Ethernet based. On the other hand, most live streaming viewers are connected to the Internet using cable or DSL networks. We run the same live streaming network in two settings: LAN-like university networks versus ADSL-like networks. We set up 20 clients, where each client is a PPLive client, with 1-second piece request time out. Clients join sequentially with a 5-second inter-arrival time; the testing lasts 100 seconds after all peers have joined. Each client's upload capacity is equal to the streaming rate of 400 Kbps, and the CDN server provides an upload capacity of 2,400 Kbps.

	LAN	ADSL
% Piece missing ratio	1.5%	7.3%
# Timed-out requests	1404.25	2548.25
# Received duplicate pkts	0	633
# Received outdated pkts	5.65	154.20

Table 3: Different results at LAN vs ADSL.

Table 3 shows the results. We observe a large performance difference between the LAN-based and ADSL-based experiments: the piece missing ratio of ADSL-based is almost 5 times that of LAN-based. Our investigation shows that the ADSL-based experiment performs worse because of large hidden buffers that are not present in LAN-like networks [23]. As an ADSL modem has a large buffer but limited upload bandwidth, a large queue can be built up at the modem and the queueing delay can reach several seconds. On the other hand, the default piece request time out value is 1 second, due to testing in the LAN-based network. This shows that testing in an environment that is different from the real setting can give misleading results and yield inappropriate system configurations.

Note that once it is determined that this is an important feature, hidden buffers can be approximated by a FIFO queue using the Linux tc tool. other network features, such as PowerBoost [2], may involve internal and dynamic ISP-policies, and hence can be much harder to emulate [14].

3. SYSTEM COMPONENTS

ShadowStream makes two extensions to a traditional live streaming network: (1) it introduces a lightweight experiment orchestrator, which extends the functions of a typical tracker to coordinate experiments; (2) it introduces new capabilities in traditional streaming clients to support live testing. ShadowStream complements analysis, simulation, and testbed to provide a more complete experimentation framework.

Figure 1 shows the major components of a ShadowStream network, their interactions, and tools such as deterministic replay that we build on top of basic ShadowStream. For scalability, the main intelligence of ShadowStream is in individual ShadowStream clients.

4. STREAMING CLIENT DESIGN

We start with the design of ShadowStream clients. We assume that experimental algorithms have been first tested in lab, going through unit and regression tests to fix correctness errors and pro-

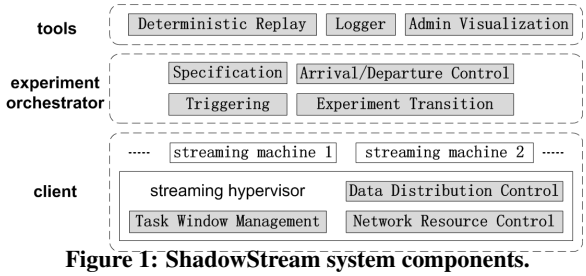

Figure 1: ShadowStream system components.

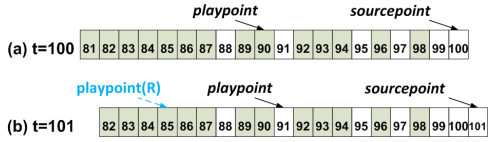

Figure 2: Streaming machine buffer: (a) at t=100; (b) at t=101.

gram crashes. Our focus is on performance. We focus on the main ideas in this section and choose to present our design in steps to make understanding easier. Implementation is in the next section.

4.1 Streaming Machine

We adopt the general, widely used piece based streaming structure, where a live streaming client downloads and uploads streaming data in units of pieces. For simplicity of explanation, we assume that each piece contains 1-second of streaming data.

We refer to a self-complete set of algorithms to download and upload pieces as a *streaming machine* or a machine for short. In a traditional setting, a streaming client of a viewer has one streaming machine. In ShadowStream, as we will see in this section, there can be multiple streaming machines executing concurrently. We refer to the streaming machine containing the experimental algorithms as `experiment`.

A key data structure of a streaming machine is its play buffer, which keeps track of pieces that are already downloaded as well as the pieces that the machine needs to download from its neighbors or CDN networks in order to feed a media player, which displays to a viewer. Figure 2 (a) is an example illustrating the play buffer status at time $t = 100$ of a client i. A shaded piece is one that has been downloaded. We follow a convention that the index of a piece is the time that the piece was produced at the source.

In a traditional setting without ShadowStream, the right most piece of play buffer is the most recent piece produced by the source. We refer to this piece as the *sourcepoint*. We also use sourcepoint to refer to the index of the sourcepoint piece. For the example in Figure 2 (a), the sourcepoint is 100. A client may not always communicate with the source to directly know about the sourcepoint. But during initialization, the client learns the time at the source and hence can always compute the time at the source through its local clock and hence the current sourcepoint. Another important piece at an instance of time is the next piece to be delivered to the media player at the client. We refer to it as the *playpoint*. For the example in Figure 2 (a), the playpoint is 90.

The time difference between the sourcepoint and the playpoint represents the *lag*, which is the delay from the time that a piece becomes available to the time that it should become visible to a viewer. A streaming machine may choose a lag when it starts. But as soon as it starts playback to a real viewer, it can no longer change the lag without disturbing the real viewer. The lag is essentially the total budget of time to finish downloading each piece. Note that the play buffer keeps some pieces older than the playpoint so that the client can serve other peers in a P2P mode.

The playpoint and sourcepoint advance in time. Figure 2 (b) shows the advancement of the playbuffer from Figure 2 (a). We see that at the next time $t = 101$, the playpoint becomes 91, and

the sourcepoint becomes 101. If the client is not able to download piece 91 from time $t = 100$ to $t = 101$, then we say that piece 91 is missing.

We comment that one may consider the piece-based streaming machine model as a quite general program model, where each piece is a computational task, and a sequence of tasks arrive at a streaming machine to be computed and the results are revealed to real viewers. Hence, we also refer to each piece to be downloaded as a task; a piece missing its deadline as a task failure.

4.2 Starting Point: R+E to Mask Failures

The starting design of ShadowStream is simple and intuitive. To reveal the true performance of `experiment`, ShadowStream assigns a task first to `experiment` (E) as if it were running alone. To protect real viewers' quality-of-experience (QoE), ShadowStream assigns a failed task to another streaming machine `repair` (R) for repair; that is, if a piece is not received by its target playback deadline, ShadowStream records a performance failure event and the responsibility for downloading the piece is shifted to R. We refer to such a design as a R+E design pattern which is shown in Figure 3.

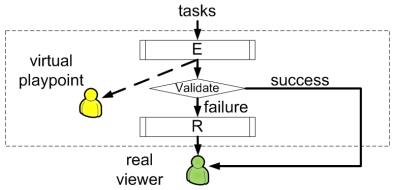

Figure 3: The general R+E design pattern.

Assume that Figure 2 (b) is the playbuffer state of a streaming machine `experiment`. Since `experiment` is not able to download piece 91, ShadowStream records that `experiment` has missed piece 91, and then assigns it to R. Assume that R is able to download piece 91 within one second. Then the piece becomes available at time $t = 102$.

To ensure that the failure of `experiment` on finishing piece 91 is not visible to the real viewer during our evaluation, we need that the playpoint of `experiment` is not the actual viewer visible playpoint. Hence, we say that the playpoint of an experimental streaming machine is a *virtual playpoint*. Specifically when $t = 101$, if the viewer visible playpoint actually is 85, then the real viewer will not see a missing piece 91.

In other words, in the R+E design, `experiment` is evaluated by its ability to finish each piece within its designed target deadline, which is 10 = (100-90) in this example, when `experiment` is running alone. But the R+E design delays the delivery of results to real viewers with an additional 5-second lag, to give R the time to repair errors, if any. We comment that the technique of introducing a slight delay in revealing results has been used in other contexts. In particular, radio and television broadcast introduces a 7-second profanity delay to handle profanity, bloopers, violence, or other undesirable material [9]. We are the first to introduce this technique to achieve our new goal of live testing.

4.3 R=`production` for Staging/Protection

A question that we have not answered is how to design R. One simple approach is that R is a machine which can check the status of each piece and download missing pieces from dedicated CDN resources provided for testing; we call this design R=`rCDN`. However, there can be limitations.

First, using R=`rCDN` requires dedicated CDN resources. Note that the dedicated CDN resources are independent of the CDN resources in other streaming machines. Suppose that a testing channel consists of 100,000 clients at a 1 Mbps streaming rate. To guarantee performance during the worst case when `experiment` cannot finish the majority of the tasks assigned to it, R=`rCDN` needs to reserve a total of 100 Gbps capacity from the CDN network. For a streaming technology company with multiple groups developing different algorithms in parallel, multiple testing channels might be running concurrently, hence the amount of reserved capacity on CDN could be too high.

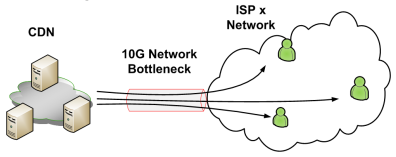

Figure 4: Network bottleneck between CDN and clients.

Second, R=`rCDN` may not work well in some network settings. Figure 4 shows a network bottleneck (*e.g.*, enterprise ingress) from the locations of the CDN edge servers to a group of streaming clients [24]. In this setting, even if the CDN has a large reserved capacity (*e.g.*, 100 Gbps), due to the bottleneck (*e.g.*, 10 Gbps), the repair capability of the CDN edge servers cannot be fully utilized.

To better understand the limitations of R=`rCDN`, we compare `rCDN` with the production streaming engine named `production`. We observe that `production` typically has more sophisticated, fine-tuned algorithms (*e.g.*, hybrid architecture) compared with a repair-only `rCDN`; `production` can use a much larger resource pool (the whole production infrastructure) than the dedicated CDN.

Hence, using R=`production` can lead to much more scalable protection. Further, it leads us to a unified approach to handle both protection and orchestration (*i.e.*, serving a client before `experiment` starts). Specifically, a ShadowStream client always uses `production` as the streaming machine to start. We can consider this state as a testing state with an `experiment` streaming machine that misses every task (*i.e.*, a no-op). When a client joins a test, a corresponding `experiment` starts. Hence, we can (conceptually) consider that a client is always in a testing phase. We refer to this as *pre-reservation* for testing. The virtual "arrival" and "departure" of an experimental streaming machine `experiment` are simple state operations. Figure 5 illustrates the transitions of states, and these virtual transitions are invisible to real viewers.

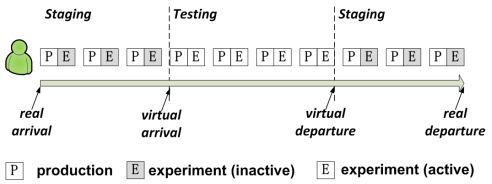

Figure 5: ShadowStream orchestration timeline: real arrival, virtual arrival, virtual departure, real departure.

4.4 Problems of R=`production`

One issue of using R=`production`, however, is that it has a systematic bias in underestimating the performance of `experiment`, due to competition between `experiment` and `production` on the shared resources: peer upload bandwidth. To protect viewers' QoE, it is desirable to give `production` a higher priority over `experiment` on using the shared upload bandwidth. However, the reduction of resources to `experiment` may lead to a systematic underestimation on the performance of `experiment`, leading a developer to unnecessarily reject `experiment` as poor performing. Below we provide a more detailed analysis on the bias. Readers who seek a solution directly can skip to the next subsection.

For a reasonably designed streaming machine, piece missing ratio $m(\theta)$ is a non-increasing function of the supply ratio θ. The curve labeled as *performance curve* in Figure 6 is an example curve of the piece missing ratio function of a typical live streaming machine. Then our goal of evaluating `experiment` is to get an ac-

curate estimation of $m(\theta_0)$ for `experiment`, during a typical live testing with supply ratio θ_0.

However, competition between `production` and `experiment` may not allow us to observe $m(\theta_0)$. Instead, the dynamic competition between these two streaming machines will lead to a *stationary point* labeled as "misleading result" in Figure 6. The point is the intersection between the performance curve and the curve representing the points whose x and y-axis values sum to θ_0. We label the x-axis (effective supply ratio to `experiment`) of the "misleading result" point as θ^*.

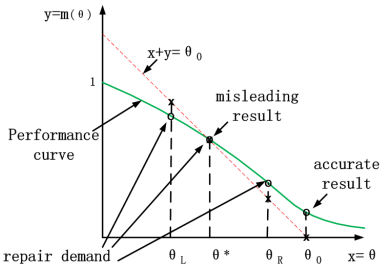

Figure 6: Systematic bias when using `production`.

To understand why θ^* is a stationary point that will be observed, first consider a supply ratio θ_R on the right of θ^*. Assume that at an instance of time the supply ratio of `experiment` is θ_R, the piece missing ratio of `experiment` will be $m(\theta_R)$. Hence, `production` takes away $m(\theta_R)$ resources to repair the losses of `experiment`. For simplicity, we assume that `production` has a repair efficiency η of 1; that is, it uses one unit of bandwidth resource to repair one unit of task. The available resources to `experiment` after the deduction become $\theta_0 - m(\theta_R)$. Since $m(\theta_R)$ is higher than $\theta_0 - \theta_R$, as one can verify by noticing from Figure 6 that the $x + y = \theta_0$ curve is a 45 degree line, we know that the remaining resources to `experiment` will become lower than θ_R $(= \theta_0 - (\theta_0 - \theta_R))$. In other words, the supply ratio of `experiment` moves from θ_R toward θ^*. One can make a similar observation when considering a supply ratio θ_L on the left of θ^*. Specifically, for a fixed η, we can conclude that θ^* satisfies:

$$\theta^* + \eta * m(\theta^*) = \theta_0.$$

In more complex settings, the efficiency of η could be a function of the specific missing pieces of `experiment` as well as the network supply ratio. Hence, one may solve for θ^* through the equation:

$$\theta^* + \eta[(m(\theta^*)] * m(\theta^*) = \theta_0.$$

To summarize, we know that the experimental results reveal $m(\theta^*)$, instead of the true $m(\theta_0)$.

4.5 Putting It Together: PCE

To remove the systematic bias of R=`production` and still provide scalable protection, ShadowStream introduces a simple, novel streaming system structure where R consists of two streaming machines: a repair CDN (referred to as `rCDN`) with bounded, dedicated repair resources and `production`. We refer to this scheme as the PCE scheme (*i.e.*, R=P+C).

Specifically, in this scheme, if `experiment` cannot download a piece by its deadline, responsibility for downloading the piece is shifted to `rCDN`. However, the dedicated CDN repair resources are only up to δ percentage of total demand, where δ is a control parameter depending on available CDN resources and the desire to obtain accurate results for a test. If a piece cannot be repaired by `rCDN`, due to either repair CDN capacity exhaustion or network bottlenecks, responsibility for downloading the piece is shifted to `production` as a final protection.

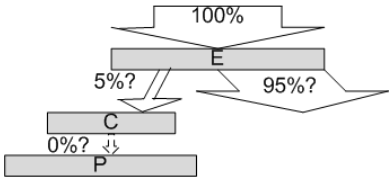

Figure 7: ShadowStream tasks processing percentages to PCE.

One way to understand the essence of the R=P+C design is to think of it as inserting a small `rCDN` "filter" between `experiment` and `production` to handle pieces missed by `experiment`. Figure 7 shows the flow. If the amount of missing pieces is lower than the capacity of `rCDN` (*e.g.*, lower than 5%), `rCDN` absorbs all leaked load of `experiment`, so that `production` is idle to avoid interference with `experiment`. On the other hand, if `experiment` has a major failure, repair load is handled by both `rCDN` and `production`, providing a guarantee to viewers' QoE.

Another way to understand the R=P+C design is that it "lowers" the piece missing ratio curve of `experiment` visible by `production` down by δ. The solid line in Figure 8 shows the actual missing ratio curve and the dashed line the lowered one. If $m(\theta_0) < \delta$, then all missing data can be retrieved by `rCDN`. Hence, `production` is idle and there is no competition on peers' upload bandwidth, resulting in an accurate experimental result. On the other hand, if $m(\theta_0) > \delta$, then `rCDN` cannot repair some missing pieces. The remaining missing pieces "overflow" to `production`.

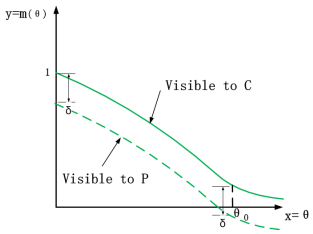

Figure 8: ShadowStream PCE design intuition.

We make a precise statement on the R=P+C design:

PROPOSITION 1. *Assume that the piece missing ratio of* `experiment` *is a non-increasing function of supply ratio. With an unknown θ value, PCE provides a bi-modal evaluation result: if $m(\theta) \leq \delta$, precise experiment accuracy can be achieved; otherwise, the system converges to a stationary point where the upper bound of $m(\theta)$ can be obtained, and we get an exact indication that $m(\theta) > \delta$.*

4.6 Extension: Dynamic Streaming

There are multiple ways to extend PCE. In particular, we present an extension of ShadowStream to dynamic streaming (*i.e.*, dynamic bitrate switching), which presents a larger design space for ShadowStream. We consider two perspectives. First, `experiment` may use dynamic streaming, and dynamic streaming experiments may be more sensitive to interference caused by protection (*i.e.*, `experiment` switching streaming rates). Second, ShadowStream protection can take advantage of dynamic streaming, which offers a new dimension for designing the protection scheme. Consider the case that both `experiment` and protection use dynamic streaming. A straightforward protection strategy is *Follow*: when ShadowStream detects that the pieces at time t is incomplete, it repairs the video for t, according to the encoding rate decided by `experiment`. However, the general idea of ShadowStream allows us to repair a different encoding rate, as long as it protects the viewer's QoE. One strategy is *Base*, which is different from *Follow* in that it always repairs the base (lowest) encoding rate. Furthermore, ShadowStream can use *Adaptive*, in which it considers both available protection bandwidth and the amount of remaining tasks at each rate left by `experiment`, and then picks the best repair.

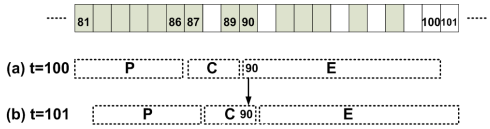

Figure 9: Streaming machine sliding download window: (a) at t=100; (b) at t=101.

5. CLIENT IMPLEMENTATION

There are challenges in implementing the novel PCE scheme designed in the preceding section. In particular, the PCE scheme consists of multiple streaming machines that may interact with each other. It is highly desirable that each streaming machine be a modular process, isolated from other streaming machines, without knowing whether it is in a test as `experiment` or `production`, or it is just running alone as `production`. The benefit of such isolation is that the production streaming machine performs the same when it has been evaluated.

A key observation of implementing ShadowStream is that we can implement the PCE scheme naturally, seamlessly, using a simple sliding window to partition the downloading tasks at a client. Based on the observation, ShadowStream imposes a minimal sliding window abstraction on streaming machines and introduces a simple *streaming hypervisor* to implement PCE.

5.1 Basic Idea

The objective of PCE is to assign a task first to `experiment`; if `experiment` fails, reassign the task to `rCDN`; if `rCDN` fails, reassign the task to `production`. We make an observation on this PCE behavior.

Specifically, at any instance of time, looking at the pieces which each streaming machine is responsible for, we observe that the streaming machines essentially partition the total downloading range spanning from the viewer playpoint to sourcepoint, with `experiment` responsible for a range of pieces that are adjacent to sourcepoint, and `production` responsible for a range that is adjacent to the viewer visible playpoint. Figure 9 (a) illustrates the partition at time $t = 100$. We call each partition the *task window* of the corresponding streaming machine.

Consider a simple scheme that we fix the size of each partition and slide each partition synchronously to the right at the playback speed. After each window movement, the oldest piece in the task window of `experiment` moves as the newest of the task window of `rCDN`. Figure 9 (b) shows that piece 90 moves from `experiment` to `rCDN`. If a piece that moves from `experiment` to `rCDN` has not been downloaded by `experiment`, it becomes a task to be finished by `rCDN`. In other words, using the simple sliding task window scheme, we achieve automatic shifting of results and failures from one streaming machine to the next in line!

The preceding observation leads to a simple, natural, modular abstraction of the key state of a streaming machine implementing the PCE scheme. Specifically, at an instance of time t, each streaming machine x is responsible for a subrange $[x_{left}(t), x_{right}(t)]$ of the total downloading range.

We introduce a streaming hypervisor to manage the PCE machines. Specifically, the streaming hypervisor provides four key sets of functions: (1) task window management (TWM) sets up sliding window; (2) data distribution control (DDC) copies data among streaming machines; (3) network resource control (NRC) conducts bandwidth scheduling among flows of streaming machines; and (4) experiment transition (ET) starts or stops experiments.

Table 4 lists key API functions between a streaming machine and the streaming hypervisor. We divide the APIs into four categories corresponding to the four sets of functions.

5.2 Task Window Management

Task window management informs a streaming machine about the pieces that it should download. We set the length of the task window of a streaming machine x as the max lag when the streaming machine is running alone:

$$x_{lag} = x.getMaxLag().$$

Hence, we focus on the right border of a task window. First consider `experiment`. We observe that $x_{right}(t)$ is actually equal to the sourcepoint t known to x. Hence, instead of notifying `experiment` $x_{right}(t)$, we notify it of the sourcepoint t. In other words, in our implementation, we provide a `getSourceTime()` function which each streaming machine can call to get the sourcepoint. For `experiment`, it is the real sourcepoint, but for `production`, it is actually the real sourcepoint minus $e_{lag} + c_{lag}$, where e_{lag} is the max lag of `experiment`, and c_{lag} is the max lag of `rCDN`. In other words, it is a "virtual sourcepoint". Notifying the right border of a task window through sourcepoint is highly desirable, because we have observed that it is a common practice for a streaming machine to define program behaviors in reference to the sourcepoint. For example, a streaming machine may move its playpoint forward to be closer to the sourcepoint after falling behind due to freezing.

Using the functions to manage the task windows is straightforward. When the streaming hypervisor is notified of a test, it gets the lag values p_{lag}, c_{lag} and e_{lag} for `production`, `rCDN` and `experiment` respectively. The start time and returned values of `getSourceTime()` are listed in Table 5.

machine	start time	`getSourceTime()`
`production`	Viewer arrival	sourcepoint - $(c_{lag} + e_{lag})$
`rCDN`	Enter testing	sourcepoint - e_{lag}
`experiment`	Enter testing	sourcepoint

Table 5: Results of calling `getSourceTime()`.

5.3 Data Distribution Control

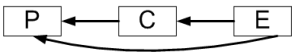

Figure 10: Directional data propagation.

Data distribution control copies pieces among streaming machines. Figure 10 shows the direction of data flows among the three streaming machines. For example, we see that a piece downloaded by C or P should not be made available to E because E may redistribute it by uploading to others, leading to misleading experimental results. In other words, the propagation of a downloaded piece is directional.

A straightforward implementation of directional data propagation is that each streaming machine has its own data store. When an earlier machine finishes downloading a piece to its data store, the piece is also copied to the data stores of those streaming machines after it. In ShadowStream, we implement a shared data store, where a piece is stored only once, for efficiency. The data store of an individual streaming machine stores pointers to the actual data pieces in the shared data store. Figure 11 shows the states of the data stores of `experiment` and `rCDN`. For example, we observe that `rCDN` has a pointer to piece 88 downloaded by itself, but `experiment` does not due to directional data distribution control.

Two functions implementing directional data control and data sharing are `writePiece()` and `checkPiece()`. A streaming machine calls `deliverPiece()` to notify the hypervisor that a piece is ready to be delivered to player; the hypervisor decides when to actually deliver the piece.

5.4 Network Resource Control

Network resource control handles the network bandwidth allocation among streaming machines, to achieve the design objectives

Call	Category	Direction	Description
`getMaxLag()`	TWM	$H \mapsto M$	Streaming machine notifies ShadowStream its maximum lag, which defines the size of its subrange.
`getSourceTime()`	''	$M \mapsto H$	Streaming machine obtains its virtual sourcepoint, which defines the right border of its subrange.
`writePiece()`	DDC	$M \mapsto H$	Streaming machine notifies ShadowStream the successful download of a piece. It updates the data stores of streaming machines that should see the piece.
`checkPiece()`	''	$M \mapsto H$	Streaming machine obtains the success/failure status of a piece.
`deliverPiece()`	''	$M \mapsto H$	Streaming machine notifies that a piece is ready to be delivered to player.
`sendMessage()`	NRC	$M \mapsto H$	Streaming machine sends a protocol/data message. ShadowStream internally manages the priority queues and may drop the message.
`recvMessage()`	''	$M \mapsto H$	Streaming machine obtains the next received message.
`start()`	ET	$H \mapsto M$	Streaming hypervisor notifies the streaming machines to activate operations.
`stop()`	''	$H \mapsto M$	Streaming hypervisor notifies the streaming machines to deactivate operations.
`rejoin()`	''	$H \mapsto M$	Streaming hypervisor notifies the streaming machines to disconnect all neighbors and rejoin.
`fetchCDN()`	''	$M \mapsto H$	Streaming hypervisor and `rCDN` directly fetch pieces from dedicated CDN resources.

Table 4: APIs (H hypervisor, M machine): $H \mapsto M$ APIs are implemented by M and called by H; $M \mapsto H$ vice visa.

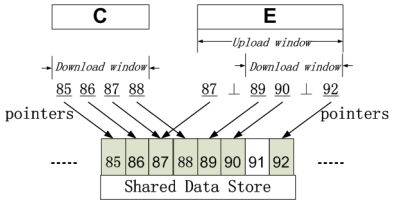

Figure 11: Using data store for data distribution control.

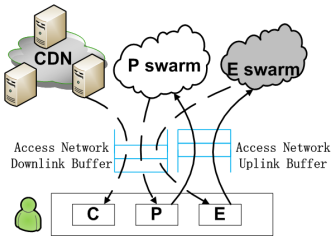

Figure 12: Network resource control

of ShadowStream. At any instance of time during a test when `experiment`, `rCDN`, and `production` are running concurrently, there are multiple sets of data flows originating from or terminating at a streaming client. Figure 12 shows 3 sets of downlink flows and 2 sets of uplink flows. In particular, consider the two sets of uplink flows due to `experiment` and `production` respectively. These two sets may compete on the shared network uplink when `experiment` performs poorly, as we have analyzed in Section 3.

Network resource control in ShadowStream assigns flows from `production` with a higher priority over those from `experiment`. However, simple priority is not sufficient as a misbehaving `experiment` may send at a high rate to create external congestions (*e.g.*, due to hidden buffers). Hence, ShadowStream resource control imposes pacing, based on an algorithm similar to LEDBAT [34] to perform dynamic bandwidth estimation.

No streaming machine can exceed estimated network bandwidth to create hidden network congestions. This feature has been implemented for both the UDP mode and the TCP mode. The provided APIs are `sendMessage()` and `recvMessage()`.

6. EXPERIMENT ORCHESTRATION

We have covered how a streaming machine can be started inside a streaming client. A capability still missing in an experimental platform is to compute the start and stop times of `experiment` in each client to create desired testing client behavior scenarios. Furthermore, when a client joins a test and changes from normal state to testing state, a smooth transition is needed to guarantee that the accuracy of the testing is not impaired.

6.1 Specification and Triggering

ShadowStream allows intuitive specification of testing client behavior patterns. Specifically, a testing behavior pattern defines one or multiple *classes* of clients, where the class of a client is defined by its properties (*e.g.*, cable or DSL, estimated upload capacity class, production software version or network location [44]).

For each class j, a behavior pattern defines:

- A class-wide arrival rate function $\lambda_j(t)$ during the interval $[0, t_{\exp}]$, where $t_{\exp}$ is the duration of the testing. For example, to generate a flash-crowd arrival in class j, $\lambda_j(t)$ will have high values initially, and then decreases sharply.
- A client life duration function L is specified by the probability that a client stays less than a duration in the testing. For live streaming, it is important that client lifetime is dependent on the arrival time [38] and video quality [28, 38]. For example, if the streaming quality at a client is below a threshold (*e.g.*, piece missing ratio $> 3\%$), the client departs. Specifically, for a client arriving at time x after experiment starts, the client's lifetime is determined by both L_x and the streaming quality.

Given a specified client behavior pattern, experiment orchestration provides two functions:

- Triggering: The experiment orchestrator monitors the testing channel to wait for the network to evolve to a state where the testing client behavior pattern can be triggered. The triggered test starting time is referred to as t_{start}. Thus, test will run from t_{start} to $t_{start} + t_{exp}$.
- Arrival/Departure Control: The orchestrator selects clients to join and depart from the testing channel to create the target testing behavior pattern.

Triggering Condition: To simplify presentation, we assume a single global client class with arrival rate function $\lambda(t)$. If an experiment is triggered at time 0 and should continue for a duration of t_{exp}, then for t between 0 and t_{exp}, an upper bound of the expected number of clients active in the experiment, denoted as $\exp(t)$, is as follows:

$$\exp(\mathrm{t}) = \Lambda(t) - \int_0^t \lambda(x)\Pr\{L_x < t - x\}\, dx, \qquad (1)$$

where $\Lambda(t) = \int_0^t \lambda(x)\, dx$. The expression is an upper bound because clients may depart due to insufficient streaming quality, which is hard to direct model. Such departures only decrease the number of active clients in the experiment.

At any time t_0, the orchestrator predicts the number of active clients in the testing channel in the interval $[t_0, t_0 + t_{exp}]$. Let $\mathrm{predict}(t)$ be the predicted value at time $t \in [t_0, t_0 + t_{exp}]$. To compute $\mathrm{predict}(t)$, the orchestrator uses a simple extension of the

autoregressive integrated moving average (ARIMA) method [43] that uses both recent testing channel states and the past history of the same program. To obtain current testing channel states, the orchestrator gathers channel state (arrivals and departures) from clients' low cost UDP reports. Specifically, at current time t_0, the orchestrator checks the condition:

$$\exp(t) \leq \text{predict}(t_0 + \Delta + t') \, \forall \, t' \in [0, t_{\text{exp}}], \qquad (2)$$

where Δ is triggering delay. If the condition is satisfied, $t_0 + \Delta$ can be triggered as t_{start}.

6.2 Independent Arrivals Achieving Global Arrival Pattern

After the triggering condition, the orchestrator needs a mechanism to notify a large number of clients in real-time about their time to join and leave a testing scenario to create a specified testing behavior pattern.

A direct approach is that one or more orchestrators issue commands at appropriate times to each client to let it join or leave the test. This approach is taken by some experimental platforms (*e.g.*, [26, 37, 41]).

However, this approach is not scalable. Furthermore, since the connections between the orchestrator and the clients can be asymmetric (*e.g.*, many real viewers in the Internet are behind NAT devices [19]), it is easier for clients to send to the orchestrator than from the orchestrator to the clients.

Hence, ShadowStream introduces distributed orchestration, which decouples the scenario parameters from their execution, and thus relaxes the requirement that control messages are delivered to each client within a small delay, in contrast with direct control.

Specifically, in the distributed control mode, after computing t_{start}, the orchestrator embeds network-wide common parameters, including t_{start}, t_{exp}, and $\lambda(t)$, into keep-alive response messages and distributes to all clients. The orchestrator does not compute or control the arrival times of individual clients. It may first appear that achieving a defined testing client behavior such as flash-crowd requires global computation of client arrival times. However, we will see that there exists an effective, distributed algorithm where each client can *locally* decide and control its own arrival times. The key is the following theorem from Cox and Lewis [13]:

THEOREM 1. *Let $T_1, T_2, \ldots$ be random variables representing the event times of a non-homogeneous Poisson process with expectation function $\Lambda(t) = \int_0^t \lambda(x)\,dx$, and let N_t represent the total number of events occurring before time t in the process. Then, conditioned on the number of events $N_{t_{\text{exp}}} = n$, the event times $T_1, T_2, \ldots, T_n$ are distributed as order statistics from a sample with distribution function $F(t) = \frac{\Lambda(t)}{\Lambda(t_{\text{exp}})}$ for $t \in [0, t_{\text{exp}}]$.*

An implication of Theorem 1 is that we can generate arrival times by drawing random numbers *independently* according to the same distribution function $F(t)$. Sorting these independent arrival times, we obtain the arrival times of clients following the desired arrival rate function $\lambda(t)$.

One remaining issue in a large-scale experiment is that the preceding theorem requires selecting a fixed n. Enforcing that exactly n clients is chosen can force the orchestrator to keep hard state about specific clients. To reduce orchestrator overhead, the orchestrator draws the total number of clients from a distribution $\hat{N}_{t_{\text{exp}}}$ with the same mean ($\text{E}[\hat{N}_{t_{\text{exp}}}] = \text{E}[N_{t_{\text{exp}}}]$) but without increasing the variance ($\text{Var}[\hat{N}_{t_{\text{exp}}}] \leq \text{Var}[N_{t_{\text{exp}}}]$). This mode trades slight variation in arrival rate for higher scalability. It permits the variance on number of total clients to be reduced in the interest of tighter control over the number of clients.

Specifically, the orchestrator computes p as a ratio of the expected value of clients in the scenario ($\Lambda(t_{\text{exp}}) = \text{E}[N_{t_{\text{exp}}}]$) to the total number of available clients M. Each client i independently participates in the scenario with probability p, and computes an arrival time $a_{e,i}$ at which it will become active in the test. This leads to a simple distributed algorithm shown in Figure 13.

Orchestrator:
01. Let M be the total number of available clients
02. Let $p = \frac{\Lambda(t_{\text{exp}})}{M}$
03. Send t_{start}, t_{exp}, $\lambda(t)$, and p to each client

Client i, upon receiving t_{start}, t_{exp}, $\lambda(t)$, and p:
04. **if** $random() > p$ **then return**
05. Draw waiting time w_i according to $F(t) = \frac{\Lambda(t)}{\Lambda(t_{\text{exp}})}$
06. Compute arrival time: $a_{\text{e},i} = t_{\text{start}} + w_i$

Figure 13: Algorithm with decentralized control for each client i to choose arrival time $a_{\text{e},i}$.

6.3 Experiment Transition

Suppose that the current time is t_0 and client i has computed according to the preceding section that it will (virtually) join the test at $a_{e,i}$. The preparation during $[t_0, a_{e,i}]$ to start `experiment` at client i consists of the following two simple steps.

Connectivity Transition: At $a_{e,i}$ when `experiment` starts at client i, one possibility that may cause unnecessary interference to `experiment` is that `production` uploads to a neighbor's `production` while the neighbor is not in test. A simple, clean solution is that during $[t_0, a_{e,i}]$, client i's `production` rejoins the testing channel, and the orchestrator returns only peers who are also in testing as neighbors to client i's `production`.

Playbuffer State Transition: Look at the target playbuffer state at time $a_{e,i}$ when `experiment` starts at client i. Figure 14 (a) gives an illustration. We have two observations: (1) pieces at the range for `experiment` should be empty; and (2) pieces for `production` and `rCDN` should be full. If these buffers are not full, `production` will download those pieces, causing interference to `experiment`. We refer to filling up `production` and `rCDN` windows directly from CDN before time $a_{e,i}$ as "legacy" removals.

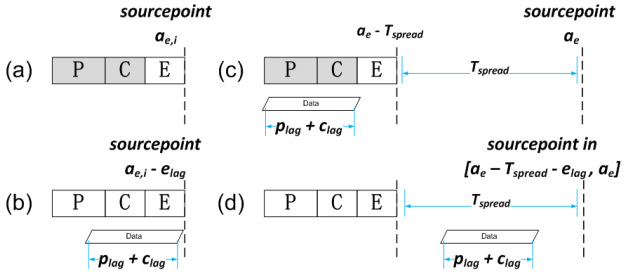

Figure 14: Playbuffer state transition.

Specifically, at time $a_{e,i}$, the piece range for `production` and `rCDN` together is $[a_{e,i} - p_{lag} - c_{lag} - e_{lag}, a_{e,i} - e_{lag}]$; note that these "legacy" pieces become all available to download at time $a_{e,i} - e_{lag}$ as shown in Figure 14 (b). For simplicity, the goal of playbuffer state transition is to pre-fetch pieces in this range during time $[a_{e,i} - e_{lag}, a_{e,i}]$. If the testing has a gradual arrival pattern, the fetching of the range is spread out evenly and there is no issue.

On the other hand, for a flash-crowd arrival, the demand of "legacy" removal would in-turn cause a flash-crowd to dedicated CDN resources. Consider the extreme case of a simultaneous flash-crowd arrival of n clients at time a_e. Then the total load due to the prefetching is $n(p_{lag} + c_{lag})$. This load should be finished from time $a_e - e_{lag}$ to a_e. The bandwidth demand rate then is $\frac{n(p_{lag}+c_{lag})}{a_e-(a_e-e_{lag})} =$

$\frac{n(p_{lag}+c_{lag})}{e_{lag}}$. If this rate is higher than the available CDN capacity, the transition cannot happen smoothly.

To avoid the issue, we can shift the PCE windows further behind in the time domain. As shown in Figure 14 (c), this extra window space is defined as T_{spread}. Note that all "legacy" pieces are available for download at time $a_e - T_{spread} - e_{lag}$. The CDN load can be finished from time $a_e - T_{spread} - e_{lag}$ to a_e, as shown in in Figure 14 (d); hence the requirement is that

$$\frac{n(p_{lag} + c_{lag})}{T_{spread} + e_{lag}} \leq R_{pre},$$

where R_{pre} is available dedicated CDN capacity. In ShadowStream, the orchestrator computes T_{spread} and announces it along with other arrival parameters. Provided APIs `start()`, `stop()`, `rejoin()` and `fetchCDN()` are shown in Table 4. In this mode, the returned value of `getSourceTime()` should be revised correspondingly.

6.4 Replace Early Departed Clients

A main challenge is that real viewers have their real departure behaviors, which cannot be controlled by ShadowStream. Clients participating in a testing scenario may switch to different channels or be terminated due to viewer-initiated operations. For example, a client could have chosen to join the test at a particular calculated time, but the real viewer has quit the testing channel before that time; such clients are called *early-departed clients*.

Capturing client state: When a client departs early, it piggybacks a small state snapshot in the disconnection message which includes the scheduled arrival time of the client. When the replacement client takes over, it "reconstructs" the state of the replaced client.

Substitution: In ShadowStream, an early-departed client is replaced by another client in the current network, which is not selected for joining testing yet. The orchestrator maintains a FIFO queue D_j for early-departed clients in class j. Upon detecting an early-departed client, the orchestrator appends the client's state to D_j. The orchestrator monitors the queue size of D_j, computes a probability that each client in class j locally chooses to become a replacement candidate, and piggybacks the probabilities in control messages. If a client in class j becomes a candidate, it connects to the orchestrator. The orchestrator checks if D_j is non-empty. If so, it dequeues the first state snapshot from the queue and includes it in the reply.

6.5 Independent Departure Control

The ShadowStream departure control has limitations due to the use of real viewers. ShadowStream cannot support arbitrary client departure patterns since the departure pattern of real viewers cannot be controlled. Instead, ShadowStream requires that the "departure" behavior pattern in testing can only be "equal to" or "more frequent" than the real viewer departure pattern. We argue that this limitation is acceptable since engineers mostly care about stressful scenarios.

The design of departure control is to fill the gap between the desired departure pattern and the real viewer departure pattern. The mechanism is similar to distributed arrival control mentioned above; we omit its details due to space limitation.

7. EVALUATIONS

In this section, we evaluate multiple aspects of ShadowStream. First, we evaluate the software architecture in terms of code size and implementation experience. Then, we evaluate the unique evaluation opportunities only available in production networks. The protection and the evaluation accuracy ability of ShadowStream is evaluated in the third part. The fourth part illustrates the performance of distributed control. In the fifth part we illustrate how ShadowStream works with dynamic streaming. Finally, we evaluate an additional feature, deterministic replay.

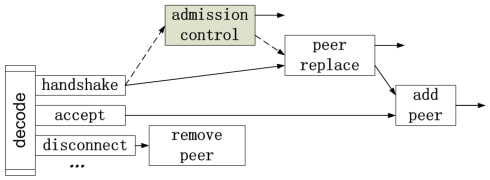

Figure 15: Adding an admission control component.

7.1 Software Framework

We implement a complete live streaming network with clients and an orchestrator using an architecture called Compositional Runtime. This block-based architecture not only supports evaluation with streaming machines and easier distribution of code for a testing scenario, but also matches well with the large distributed peer-to-peer live streaming networks, which typically consist of a set of key algorithmic components, such as connection management, upload scheduling, admission control, and enterprise coordination.

Overview: The key objective of the Compositional Runtime is that the software structure should allow modular design of algorithmic modules as well as easy composition of a system consisting of all algorithmic modules for a test. Our software architecture is inspired by prior architectures, such as Click [21], GNU Radio [17] and SEDA [40]. Algorithmic components are implemented as independent blocks that define a set of input and output ports over which messages are received and transmitted. A runtime scheduler is responsible for delivering packets between blocks.

System base: Since we have implemented the full system, we present statistics on the size of the code to illustrate that the framework is simple yet powerful. The framework and live streaming machine are implemented in C++. The full system is divided into multiple components:

- Compositional Runtime (including scheduler, dynamic loading of blocks, etc): 3400 lines of code;
- Pre-packaged blocks (HTTP integration, UDP sockets and debugging): 500 lines of code;
- Live streaming machine: 4200 lines of code.

Flexibility: Members of our research group have implemented application-layer rate limiting, modified download schedulers, and even push-based live streaming by simply writing new blocks and updating a configuration file.

Figure 15 illustrates a portion of a live streaming machine that is responsible for managing neighbor connections. A block *decode* is responsible for de-multiplexing received packets and sending them to connected blocks responsible for handling each type of message.

Next, the designer wishes to add an *admission control* algorithm to avoid reduced performance for existing clients during flash-crowd. Admission control may be implemented as an independent block, which reads handshake messages for newly-connected clients. The block emits either the handshake message if the new connection should be accepted or a disconnect message to the client if the connection should be rejected. The designer then composes the block as shown by the dotted line.

7.2 Experimental Opportunities

We demonstrate that live testing provides experiment scales that are not possible in any existing testbed. We use real traces from two live streaming testing channels, SH Sports channel and HN Satellite channel, from one of the largest live streaming networks for 1 week beginning on September 6, 2010.

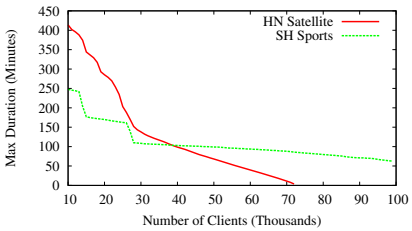

Figure 16: Experiment opportunities.

When evaluating experiment opportunities, we use a demanding testing behavior pattern: the arrival curve is a flash-crowd that all clients arrive within $\Delta = 1sec$, and no clients depart from the test. Figure 16 demonstrates the test size and test duration that can be triggered. In this figure, the x-axis is the triggered size and the y-axis is the possible duration at the size.

We make the following observations. First, the two real testing channels can allow us to test 60,000 clients for at least 40 minutes, a reasonable duration. In addition, the SH Sports channel can accommodate a test of 100,000 clients for at least 60 minutes. This scale is not possible with any testbed that we are aware of. Second, we observe that different channels can provide different tradeoffs. While the SH sports channel can provide a large channel size, the HN channel allows us to run experiments at longer durations.

7.3 Protection and Accuracy

We evaluate protection and accuracy by running fully implemented ShadowStream clients on Emulab. This is a major lacking of our evaluations, as results from production deployment may reveal unexpected issues, but production deployment of ShadowStream is currently in a work-in-progress state. On the other hand, Emulab allows us to conduct an experiment multiple times, with and without ShadowStream, all in the exact same setting. This is not possible using a production deployment. Hence, Emulab results can reveal better on ShadowStream accuracy. We plan to report production results as they become available.

Specifically, we customize a Fedora Linux image with Modelnet support for Emulab, in order to increase the number of clients. Each result is the mean value of 10 repeated evaluations.

In the test, there are 300 clients joining the testing channel with an inter-arrival time of 5 seconds. After all clients have joined, they play for 100 seconds. The channel rate is 400 Kbps and each peer's upload capacity is 500 Kbps. We use a stable P2P streaming machine as `production`. We use a standard HTTP server as the CDN server and fix the length of the `rCDN` window as 10 seconds. The capacity limitation of the CDN is set to 2.4 Mbps using the Linux tc tool. Note that for 300 clients, the average lifetime is 850 seconds; the total demand of the experiment is roughly $850*300 = 255,000$ seconds of data. The total supply of the CDN is bounded by $2400/400 * 1700 = 10,200$ seconds of data; hence the δ value is bounded by $10,200/255,000 = 4\%$.

A dedicated `experiment` version is customized for the purpose of evaluation. We inject a bug to the `experiment`, where pieces with the lowest digit equal to 1 are excluded from the download window until they enter the urgent window (*i.e.*, 3 seconds prior to the playpoint). We choose this simple setting to highlight the fact that even simple parameter changes can have a serious performance impact and therefore continuous testing is essential.

In our evaluation, we focus on two piece missing ratios for clients: (1) virtual playpoint e_{play} of the `experiment`, and (2) viewer playpoint c_{play}/p_{play} at the `rCDN`/`production`.

R=`rCDN` design: We start with getting the performance of the buggy version by running the `experiment` alone. As shown in the second column of Table 6, the buggy version yields poor performance, confirmed by the increased piece missing ratio of 8.73% when running alone. In the next test (third column in Table 6), we use R=`rCDN` and remove the CDN capacity limitation. The measured piece missing ratio is accurate with a negligible error (which we ascribe to the non-determinism the real network); viewer-observable miss at c_{play} is 0. These results are consistent with our proposition that when CDN capacity is not the bottleneck, R=`rCDN` can achieve very high experimental accuracy and viewers' QoE is guaranteed.

However, when we add a CDN limitation in the third test (forth column in Table 6), the missing ratio at c_{play} increases to 5.42%. Only around 3.4% pieces missed by the `experiment` are repaired by `rCDN`, which is less than the 4% capacity. This is because at the initial stage of testing, the number of clients in testing is small; at this stage CDN capacity can not be fully utilized. As we have stated before, when network bottlenecks exist, the viewer perceived quality could be seriously impacted.

	Buggy	R=`rCDN`	R=`rCDN` with bottleneck
e_{play} Missed	8.73%	8.72%	8.81%
c_{play} Missed	N/A	0%	5.42%

Table 6: Experiment accuracy v.s. CDN capacity.

PCE design: Next, we use our PCE design. The results are shown in the second column of Table 7. Due to the presence of a CDN bottleneck, `production` repair is activated; thus we see that the `experiment` evaluation is inaccurate. As we have expected, the network evolves to a stationary point; we get an upper bound value of missing ratio as 9.13%. A key observation is that, even when the `experiment` has poor performance, the piece missing ratio at p_{play} is only 0.15%. This test illustrates the graceful and scalable protection provided by the PCE design.

As the final test, we increase the CDN bottleneck to 9.6 Mbps. As shown in the third column of Table 7, the resulting piece missing ratio is a good approximation of the `experiment`. And again, the viewer perceived quality is guaranteed; there is no piece missing observed at p_{play}.

	PCE	PCE with higher CDN bottleneck
e_{play} Missed	9.13%	8.85%
p_{play} Missed	0.15%	0%

Table 7: PCE is more scalable and guarantee viewers' QoE.

7.4 Experiment Control

Next we evaluate experiment control. We use real client behaviors from one live streaming network captured during September 2010 to perform a trace-driven simulation. However, due to the lack of traces of viewers' departure behaviors, this part focuses on studying the distributed arrival performances. Since distributed departures perform similarly to arrivals, we believe this part is already sufficient for illustrating the effectiveness of distributed control.

Accuracy of Distributed Arrivals: We evaluate the accuracy of the distributed arrival algorithm in Section 6. Specifically, we use the arrival rate function shown in Figure 1 of [3]. When instantiated in ShadowStream, it has 349,080 arrivals. To test the impact of client clock synchronization, each client perturbs its generated arrival time by a uniform random value selected from the interval $[-\frac{s}{2}, \frac{s}{2}]$, where s is a parameter that the orchestrator announces.

First, we observe that distributed arrivals result in a close match to the target arrival rate function. The results are shown in Figure 17, which compares the target arrival rate with the actual arrival rate resulting from clients joining the channel with orchestrator. In this experiment, the orchestrator sets the clock skew parameter $s = 3$, so clients' clocks are desynchronized by up to 3 seconds.

Next, we further evaluate the effects of clock desynchronization on distributed arrivals. In particular, for varying values of s, we test

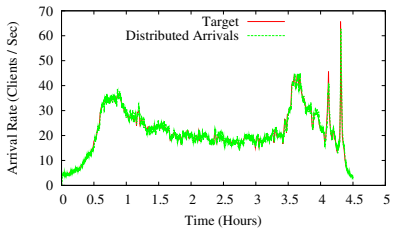

Figure 17: Experiment Control generates accurate arrival behaviors. Clients have clocks differing by up to 3 seconds.

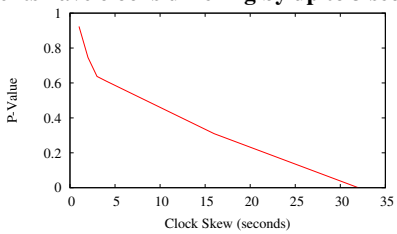

Figure 18: Experiment Control robust to desynchronization.

the hypothesis that an observed arrival behavior could have been generated by rate parameter $\lambda(t)$ using a Chi-Square test. The null hypothesis is that the arrivals in interval $[t_1, t_2)$ are distributed according to $\Lambda(t_2) - \Lambda(t_1)$. We place arrival times in an intervals of 3 seconds, and then merge adjacent intervals until both the expected number of arrivals and the number of arrivals in each are at least 5.

Figure 18 shows the results. The x-axis is the level of clock desynchronization and the y-axis is the corresponding p-value (*i.e.*, the probability that the deviation of the observed from that expected). We observe that if client clocks are synchronized within 3 seconds with the orchestrator, the distributed arrivals are accepted with high confidence (the p-value is at least 0.6); we reject the null hypothesis for clock skews $s > 20$ seconds. Synchronization to a precision of 3 seconds is readily doable: in ShadowStream, peers would use sourcepoint value from source as time reference.

7.5 Dynamic Streaming

We evaluate ShadowStream when `experiment` implements dynamic rate switching among 4 rates: 1x (400 Kbps), 2x (800 Kbps), 3x (1.2 Mbps) and 4x (1.6 Mbps). Specifically, `experiment` switches its streaming rate by measuring buffer-filling-ratio (*i.e.*, fraction of downloaded pieces) and adapts according to the following Adobe OSMF-like rule: (1) every peer starts from rate 1x (400 Kbps); every 120 seconds, it re-calculates the buffer-filling-ratio; (2) when the buffer-filling-ratio is above 80%, switch to a higher level rate (overlay) if possible; (3) when the buffer-filling-ratio is below 50%, switch to a lower level rate (overlay) if possible. The protection decision is made every 10 seconds, considering missing video pieces to be played in the next 10 seconds.

	Follow	*Base*	*Adaptive*
Accuracy	1.26x	1.26x	1.26x
Protected QoE	1.59x	1.42x	1.58x
Protection Overhead	1.49	3.69	1.39

Table 8: Evaluation of Repair schemes.

We evaluate ShadowStream performance from three perspectives: (1) Accuracy (to reflect interference by protection to `experiment`), measured by average streaming rate observed by `experiment`; (2) Protected QoE, measured by average streaming rate observed by `production`; and (3) QoE protection overhead, measured by per-client protection downloading rate (in Kbps).

We report the results of a setting of 300 clients with a supply ratio of 2.5 over the 1x rate. We first run `experiment` alone. The average downloading rate is 1.59x. However, if a segment, which contains 10-second video data, is incomplete, we do not count the segment when calculating the viewer QoE streaming rate. The average viewers' QoE streaming rate when running `experiment` alone is 1.26x.

Table 8 shows the results of using the three protection strategies discussed in Section 4.6. We make two observations. First, for the dynamic streaming `experiment`, the protection overhead is low. None of three protection strategies need more than 3.5 Kbps (<1% 1x rate) average protection downloading rate. Dynamic streaming experiments may need lower protection overhead because that they already try to do self-repair: when the buffer-filling-ratio of a client decreases, it switches to a lower rate.

Second, *Base* is not a good protection strategy. The protected QoE of *Base* is only 1.42x, while the other two strategies achieve 1.59x/1.58x. At the same time, the protection overhead of *Base* is higher than twice that of the other two. The reason is that *Base* often discards partially-downloaded pieces by `experiment` while the other two reuse them. In particular, *Adaptive* has the lowest protection overhead.

7.6 Deterministic Replay

We build a replay capability on top of ShadowStream so that real tests in production streaming can be played back step-by-step offline for detailed analysis. A main challenge for introducing this capability is how to minimize the logged data. Operating in a large-scale production environment, ShadowStream clients cannot log and then upload a large amount of data.

The hypervisor structure of ShadowStream allows us to implement efficient deterministic replay by controlling non-deterministic inputs. Specifically, a streaming machine is typically an event-driven system. Since external events are easier to log and replay, ShadowStream explicitly transforms a timer event input to an external input provided as a streaming hypervisor service. The event queue and message queue of each streaming machine are maintained by the streaming hypervisor. Random number generations always use the sourcepoint from the *getSourceTime()* call as seeds. Note that the logging implementation can be optimized: for protocol packets, we can save the whole payload; for data packets, only the packet headers are needed. Since our streaming machines are not computationally-intensive, we implement each streaming machine in a single thread. Using per-client input logs, we can replay any client's behavior.

Table 9 shows the per-client log size in two evaluations: one with 100 clients running for 650 seconds and the second with 300 clients running for 1800 seconds. The channel rate is 480 Kbps. We observe that the sizes of logged data to achieve replay are practical: only 223 KB for the first case, and 714 KB for the second.

	Log Size
100 clients; 650 sec	223 KB
300 clients; 1800 sec	714 KB

Table 9: Logged data size for deterministic replay.

8. RELATED WORK AND DISCUSSIONS

Q: [Related Work] How is ShadowStream related to previous work?
A: There are significant previous studies on the debugging and evaluation of distributed systems (*e.g.*, [4, 16, 18]). Compared with the preceding work, ShadowStream is the first system based on the key observation, instantiated in the context of live streaming, that both the `production` system and the `experiment` system can contribute to the objective of the same function: fetching streaming data. Another advantage of ShadowStream is that it allows specific scenarios during live testing, while previous studies focus on the current deployment state.

ShadowStream is also different from staging/provisioning/rollback, as used in industry. Such techniques cannot protect real viewers

from visible disruptions, and have limited capability to orchestrate evaluation conditions.

A particularly related project is FlowVisor [35], which proposes to allocate a fixed portion of tasks and resources to `experiment` to evaluate performance. The scope of their approach, however, is limited to only linear systems. On the other hand, complex systems such as live streaming networks can be highly non-linear due to factors such as network bottlenecks.

Q: [General Approach] Is the ShadowStream approach general?

A: ShadowStream's *Experiment→Validation→Repair* scheme is general and can be extended to other applications. Specifically, it requires only that (1) failures of `experiment` can be efficiently identified; (2) one can introduce effective Repair to mask `experiment` failures. For (1), many computer science problems/systems have the asymmetric property that efficient validation of a solution is feasible, but better algorithms take time to develop (*e.g.*, NP problems). For (2), ShadowStream offers flexibility on designing Repair as long as it masks obvious user visible failures. Note that the additional lag is NOT a necessity in the general scheme.

Generalizing ShadowStream to other applications should consider the specific problem domain. Consider video-on-demand (VoD). One difference between live and VoD is data availability: all pieces are available in VoD at any time and a streaming machine hence can download pieces forward to the end of the video. Also, the playpoints of VoD viewers can be heterogenous due to arrival time differences and viewer seek/pause. To apply ShadowStream for P2P VoD, a developer can impose a downloading range to avoid overlapping tasks between `experiment` and `production`.

Q: [Limitations] Are there limitations on the experiments that can be conducted by ShadowStream?

A: Yes. First, ShadowStream introduces a small additional lag. Users accept such small lags in many production channels. If the required lag is unacceptable, ShadowStream cannot embed experiments. Second, the ability of ShadowStream protection depends on the Repair mechanism to mask user-visible failures of `experiment`. An experiment cannot be conducted if Repair and `experiment` use different codecs and media player cannot use both. Third, ShadowStream assumes that production is always running and that management traffic (monitoring and logging) is small. This is typically true (*e.g.*, 1 report packet per 30 seconds), but if not, the accuracy of ShadowStream may be reduced.

Acknowledgments: We are grateful to Ion Stoica (paper shepherd), Nicole Shibley, Lili Qiu, Ye Wang, Yin Zhang, Hongqiang Harry Liu, David Wolinsky, Peng Zhang, Haiwei Xue, Junchang Wang and anonymous reviewers for valuable suggestions.

9. REFERENCES

[1] Cisco Visual Networking Index: Forecast and Methodology, 2011-2016. `www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf`.

[2] PowerBoost. `broadbandreports.com/shownews/75298`.

[3] S. Agarwal, J. P. Singh, A. Mavlankar, P. Baccichet, and B. Girod. Performance and Quality-of-Service Analysis of a Live P2P Video Multicast Session on the Internet. In *IWQoS 2008*.

[4] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *ACM SOSP 2009*.

[5] IETF ALTO. `datatracker.ietf.org/wg/alto/charter/`.

[6] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. In *SIGCOMM 2002*.

[7] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *SIGCOMM 2006*.

[8] T. Bonald, L. Massoulie, F. Mathieu, D. Perino, and A. Twigg. Epidemic Live Streaming: Optimal Performance Trade-offs. In *SIGMETRICES 2008*.

[9] Broadcast Delay. `en.wikipedia.org/wiki/Broadcast_delay`.

[10] M. Castro, P. Druschel, A. marie Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth Multicast in Cooperative Environments. In *ACM SOSP 2003*.

[11] H. Chang, S. Jamin, and W. Wang. Live Streaming Performance of the Zattoo Network. In *IMC 2009*.

[12] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an Overlay Testbed for Broad-coverage Services. *ACM SIGCOMM CCR*, 33(3):3–12, 2003.

[13] D. R. Cox and P. A. W. Lewis. *The Statistical Analysis of Series of Events*. Methuen, 1966.

[14] M. Dischinger, K. P. Gummadi, A. Haeberlen, S. Saroiu, and I. Beschastnikh. SatelliteLab: Adding Heterogeneity to Planetary-Scale Network Testbeds. In *SIGCOMM 2008*.

[15] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing Residential Broadband Networks. In *IMC 2007*.

[16] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global Comprehension for Distributed Replay. In *NSDI 2007*.

[17] GNU Radio. `www.gnu.org/software/gnuradio/`.

[18] D. Gupta, K. V. Vishwanath, and A. Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *NSDI 2008*.

[19] Y. Huang, T. Z. J. Fu, D.-M. Chiu, J. C. S. Lui, and C. Huang. Challenges, Design and Analysis of a Large-scale P2P-VoD System. In *SIGCOMM 2008*.

[20] M. S. Kim, T. Kim, Y.-J. Shin, S. S. Lam, and E. J. Powers. A Wavelet-Based Approach to Detect Shared Congestion. In *SIGCOMM 2004*.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. on Computer Systems*, 18(3):263–297, Aug. 2000.

[22] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *ACM SOSP 2003*.

[23] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the Edge Network. In *IMC 2010*.

[24] R. Krishnan, H. V. Madhyastha, S. Jain, S. Srinivasan, A. Krishnamurthy, T. Anderson, and J. Gao. Moving Beyond End-to-End Path Information to Optimize CDN Performance. In *IMC 2009*.

[25] R. Kumar, Y. Liu, and K. Ross. Stochastic Fluid Theory for P2P Streaming Systems. In *INFOCOM 2007*.

[26] L. Leonini, E. Riviere, and P. Felber. SPLAY: Distributed Systems Evaluation Made Simple. In *NSDI 2009*.

[27] B. Li, G. Y. Keung, C. Lin, J. Liu, and X. Zhang. Inside the New Coolstreaming: Principles, Measurements and Performance Implications. In *INFOCOM 2008*.

[28] Z. Liu, C. Wu, B. Li, and S. Zhao. Why Are Peers Less Stable in Unpopular P2P Streaming Channels? In *Networking 2009*.

[29] N. Magharei and R. Rejaie. PRIME: Peer-to-Peer Receiver-drIven MEsh-based Streaming. In *INFOCOM 2007*.

[30] A. Mansy and M. Ammar. Analysis of Adaptive Streaming for Hybrid CDN/P2P Live Video Systems. In *ICNP 2011*.

[31] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, A. E. Mohr, and E. E. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *IPTPS 2005*.

[32] F. Picconi and L. Massoulie. Is there a future for mesh-based live video streaming? In *P2P 2008*.

[33] PPLive User Comments. `tieba.baidu.com/f?kz=700224794`.

[34] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). IETF Draft.

[35] R. Sherwood, G. Gibb, K. kiong Yap, M. Casado, N. Mckeown, and G. Parulkar. Can the Production Network be the Testbed. In *OSDI 2010*.

[36] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescape. Broadband Internet Performance: A View From the Gateway. In *SIGCOMM 2011*.

[37] C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab Application Management Using Plush. *Operating Systems Review*, 40(1), Nov. 2006.

[38] I. Ullah, G. Bonnet, G. Doyen, and D. GaÃńti. Modeling User Behavior in P2P Live Video Streaming Systems through a Bayesian Network. In *MAMNS 2010*.

[39] V. Venkataraman, P. Francis, and J. Calandrino. Chunkyspread: Multi-tree Unstructured Peer-to-Peer Multicast. In *IPTPS 2006*.

[40] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *ACM SOSP 2001*.

[41] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI 2002*.

[42] C. Wu, B. Li, and S. Zhao. Diagnosing Network-wide P2P Live Streaming Inefficiencies. In *INFOCOM 2009*.

[43] C. Wu, B. Li, and S. Zhao. Multi-channel Live P2P Streaming: Refocusing on Servers. In *INFOCOM 2008*.

[44] H. Xie, Y. R. Yang, A. Krishnamurthy, Y. Liu, and A. Silberschatz. P4P: Provider Portal for Applications. In *SIGCOMM 2008*.

[45] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li. Design and Deployment of a Hybrid CDN-P2P System for Live Video Streaming: Experiences with LiveSky. In *ACM MultiMedia 2009*.

[46] Y. Zhou, D.-M. Chiu, and J. C. Lui. A Simple Model for Analyzing P2P Streaming Protocols. In *ICNP 2007*.