# A Transport Layer for Live Streaming in a Content Delivery Network

LEONIDAS KONTOTHANASSIS, RAMESH SITARAMAN, JOEL WEIN, DUKE HONG, ROBERT KLEINBERG, BRIAN MANCUSO, DAVID SHAW, AND DANIEL STODOLSKY

*Invited Paper*

*Streaming media on the Internet has experienced rapid growth over the last few years and will continue to increase in importance as broadband technologies and authoring tools continue to improve. As the Internet becomes an increasingly popular alternative to traditional communications media, Internet streaming will become a significant component of many content providers' communications strategies. Internet streaming, however, poses significant challenges for content providers, since it has significant distribution problems. Scalability, quality, reliability, and cost are all issues that have to be addressed in a successful streaming media offering.*

*Streaming content delivery networks (streaming CDNs) attempt to provide solutions to the bottlenecks encountered by streaming applications on the Internet. However, only a small number of them has been deployed, and little is known about the internal organization of these systems. In this paper, we discuss the design choices made during the evolution of Akamai's CDN for streaming media. In particular, we look at the design choices made to ensure the network's scalability, quality of delivered content, and reliability while keeping costs low. Performance studies conducted on the evolving system indicate that our design scores highly on all of the above categories.*

***Keywords*—***Content delivery, fault tolerance, performance, streaming.*

## I. INTRODUCTION

The Internet broadband revolution is likely to significantly change the way that we interact with computers and the Internet as a whole. Internet streaming is expected to play an increasingly important role in an online world with high-bandwidth connections. However, even when end users have high-bandwidth connections to the Internet, the problem of distributing the content to them will be a limiting factor for any content provider that wants to reach that audience.

Streaming media content delivery networks (streaming CDNs) [1], [2] attempt to address the stream distribution problem much the same way that CDNs [3], [4] address the object distribution problem for regular Hypertext Transfer Protocol (HTTP) traffic. However, there exist significant differences between the two spaces. Streaming objects are typically much larger than web objects and can impose significantly higher pressure on caches. Furthermore, live streaming does not lend itself to caching at all and requires much tighter cooperation between the producers of the content and the CDN. Finally, latency between servers and clients is less of a concern in the context of streaming than it is in the context of HTTP content delivery. This is due to the fact that stream startup times (typically 2–5 s) are relatively high compared to the latency between servers and clients, which rarely exceeds a few hundred milliseconds in the worst of cases. Given these differences, there is much debate on whether and how much the architecture of streaming CDNs should resemble that of CDNs for HTTP delivery. The most successful CDNs for HTTP delivery employ a geographically distributed structure, while in the streaming space a number of commercial CDNs have adopted more centralized solutions. Akamai Technologies, Cambridge, MA [5], has adopted the distributed approach. While we present in Section II some of the data that motivates this design choice, the goal of this paper is not to justify this decision; rather, given the choice of a distributed architecture, we describe the architectural challenges it presents, the solutions we implemented, and the results achieved, focusing almost entirely on live streaming.

While designing and implementing the network we have kept the following goals in mind: scalability, high quality, availability, and low cost. From a business perspective, a

Manuscript received October 14, 2003; revised January 30, 2004. The work of R. Kleinberg was supported in part by a Fannie and John Hertz Foundation Fellowship.

L. Kontothanassis was with Akamai Technologies, Cambridge, MA 02139 USA. He is now with HP Labs, Cambridge, MA 02139 USA.

R. Sitaraman, J. Wein, D. Hong, B. Mancuso, D. Shaw, and D. Stodolsky are with Akamai Technologies, Cambridge, MA 02139 USA.

R. Kleinberg was with Akamai Technologies, Cambridge, MA 02139 USA. He is now with the Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139 USA.

Digital Object Identifier 10.1109/JPROC.2004.832956

CDN has to score well in all four of the above categories in order to attract customers, but additional characteristics are also required. For example, customers need to: 1) maintain a sense of control over their streams; 2) be able to purge content from the network as if it was located in a single centralized server; 3) have both historical and real-time statistics on their traffic; 4) be able to control their cost structure by determining how many streams to serve; and 5) be able to effect admission control of end users to their streams. While undeniably important and covered by our CDN solution, these other characteristics of a streaming CDN are not covered in this paper.

- **Scalability**: We would like to have a system where capacity constraints can be resolved simply with additional hardware deployment. This implies that the design cannot have any inherent bottlenecks either on the number of streams that it can carry or on the popularity (i.e., number of end users) for any of the streams. In this context, we introduce a mechanism called *portsets* that allows us to virtualize the concept of a streaming CDN. Using this design, we can allocate a large number of virtual CDNs—enough to handle any anticipated streaming growth—but we can multiplex them over the same physical infrastructure to reduce cost when the additional capacity is not needed.
- **Quality**: The network should deliver quality that is equal to or better than any streaming solution based on centralized client–server architectures. The basic idea behind our approach is to *reflect* a stream across multiple intermediate locations and reassemble it on the edge of the network before transmitting it to end users. Retransmissions of lost packets and rapid transmission of early packets (a technique we call *prebursting*) allow us to improve quality even further.
- **Reliability**: Our network should have no single points of failure. Software, machine, and even WAN failures should be dealt with transparently when possible and should result in graceful degradation of service. (Obviously, delivering a stream to an end user inside a failed network is not possible.) Our approach relies on placing redundant components of our streaming CDN in multiple networks and implementing a failover scheme in the presence of failures.
- **Cost**: The network should minimize the cost of streaming delivery for customers. While this is a well-understood priority in business environments, it is often at odds with the preceding goals. It is only through detailed analysis of the system and careful engineering that all of the mentioned goals can be reconciled.

The mechanisms described in this paper are implemented on a commercial streaming CDN spanning thousands of servers and tens of countries and networks. It is integrated with the three dominant streaming formats in the marketplace today (Windows Media, Real, QuickTime) [6]–[8] and is actively serving millions of streams per day. Large streaming events are handled on a regular basis with the largest events serving more than 80 000 concurrent users, and delivering more than 16 Gb/s of concurrent streams.

The rest of this paper is organized as follows. Section II discusses the original design of our live streaming CDN. The goal of that design was the reliable delivery of a small number of highly popular events. Section III presents the design modifications necessary to eliminate all bottlenecks and achieve practically unlimited scalability using virtual CDNs. Section IV presents the mechanisms used to ensure that end-user quality is as high as possible. In particular. we present three different techniques used to enhance end-user experience: lost packet recovery, redundant delivery, and prebursting. We also quantify the performance implications of those techniques by presenting experimental data from our distributed monitoring agents. Section V discusses the types of failures possible in our system and the mechanisms we have developed to tolerate such failures. In Section VI we present a few important performance metrics that showcase the ability of the system to match our initial design goals. We present related work in Section VII and our conclusions in Section VIII.

## II. Early Content Delivery Network for Live Streaming Media

Fig. 1 depicts a high-level view of our streaming CDN as it is used in the context of live streaming; in this paper, we omit all discussion of issues arising in our CDN for on-demand streaming media. Historically, our initial focus was the design of a streaming CDN that could be used for very popular events. As can be seen from the figure, there exist two types of machines in the system: streaming servers that serve end users and transport servers that consist of entrypoints and reflectors that distribute a stream from a customer's encoder to the streaming servers.

Streaming servers are organized in groups deployed in individual data centers that are tied together by a private LAN. We often refer to such a collection of streaming servers as a *region*. Regions are deployed widely so as to provide good geographical and network coverage. They also provide fault tolerance through redundancy at many levels. Since each region contains multiple servers, the failure of any one server can be detected and traffic directed to other servers in the region. End users are directed to a particular edge region by the Akamai mapping system, which maps end users to regions to which they have good connectivity. Network failures that may isolate entire regions are also detected by mapping and traffic is redirected to alternative, better connected regions. The details of the mapping system are beyond the scope of this paper.

Our choice of a wide distribution of edge regions was partially motivated by studies we have conducted suggesting that centralized solutions, due to varying network conditions, can suffer significant outages and other problems when monitored over extended periods. Fig. 2 portrays one such experiment, in which we monitored the failure rate of a stream served from just one of our U.S.-based edge regions as opposed to that of the same stream served from our

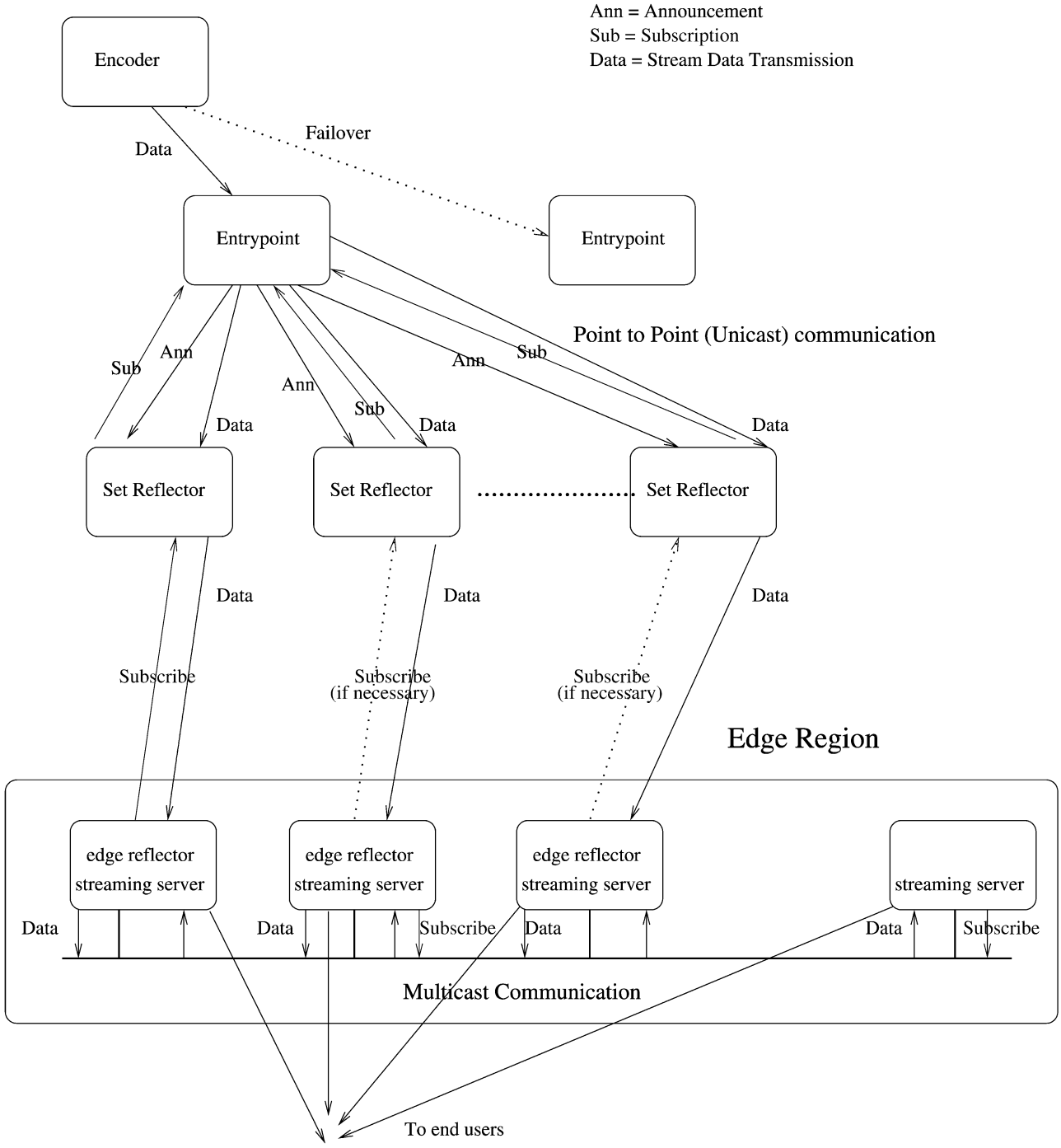


**Fig. 1.** High-level diagram of a streaming CDN.

entire distributed network. The stream was being accessed by approximately 30 agents, distributed 70% in the United States, 20% in Europe, and 10% in Asia, and the experiment covers a monitoring period of 30 days and 51 000 data points. While for many days the failure rate experienced by the centralized solution is equivalent to that of the decentralized one, there exist days during which the centralized solution exhibits a large number of failures. Those failures stem from network connectivity failures between the agents and the centralized region, since we ensured that server load was not an issue while the experiment was run.

The existence of fault-tolerant edge regions is just the first step toward a successful streaming CDN. The issue that we address in this paper is the mechanisms necessary to get live streams from an encoder to those highly distributed edge regions with good enough quality so they can then be served effectively to end users.

Streams typically originate in a customer's encoder. A collection of machines, termed *entrypoints*, acts as origin proxies of customer's encoders. Encoders are required to either connect to or accept connections from entrypoint machines and send the stream data to the entrypoints. Stream

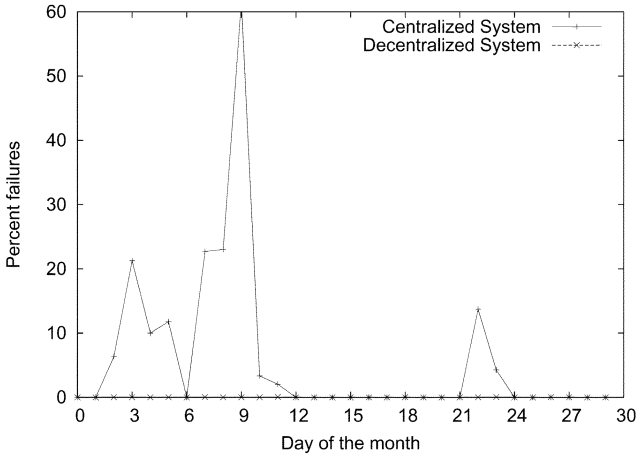


**Fig. 2.** Failure rates for centralized versus decentralized stream delivery.

data is then fanned out to a larger collection of machines termed *set reflectors*. Set reflectors in turn propagate this data to edge regions. The use of set reflectors is primarily motivated by the desire for scalability—by creating a transmission tree, we can distribute streams to much larger number of edge regions than we would have been able to otherwise. Streams arriving at a region are multicast on the private network interface shared by all streaming servers in that region, with a unique multicast address associated with every stream. All servers in an edge region can then join the appropriate multicast addresses associated with the streams that the end users are requesting, get the data for those streams from the shared back-end network, and serve it to the end users. Serving data to end users is done via unicast, since IP multicast in WANs is largely unavailable. The *entrypoint* and *set reflector* machines which are responsible for the stream transport are also known as the *reflector network*, a term that will be used throughout this paper.

This hierarchical design has several desirable properties.

- Stream data can be distributed in a scalable fashion to a large number of edge regions where it can be optimally served to end users. In addition to great scalability, this approach allows streams to be served from an edge region to which the client has good connectivity.
- Entrypoint machines can be chosen to provide good connectivity to a customer's encoder, thus ensuring good first-mile behavior.
- The intermediate layer of set reflector machines can be placed in locations with good connectivity/peering to ensure high quality delivery of the streaming data to edge regions.

From a business perspective, this design is ideally suited to serve a small number of very popular streams. The overheads associated with the transport of streams between encoders and edge regions can be amortized over a large number of end users. Furthermore, dealing with a small number of very popular streams has significantly less operational risk, since encoders and entrypoints can be under human supervision for fault tolerance purposes. Our goal, however, was broader than this design could meet. We were interested in scalability both across popular and unpopular streams, low costs of data transport regardless of stream popularity, and low packet loss across our transport layer. We explain these issues and our solutions to them in detail in the coming sections.

## III. Scalability Enhancements

Scalability limitations of the first design were detected early on as the amount of traffic on the system increased. While our initial design was targeted at highly popular live streams, we discovered that at any point in time many streams were not very popular. Delivering those streams to edge regions where they were not needed overloaded the set reflectors and increased the cost of the system unnecessarily. Solving this problem required the development of a subscription system, which is described in Section III-A.

Even after the development of a subscription system, we discovered that certain set reflector machines could get overloaded due to large load imbalances. Commonly, certain edge regions would become very active and request a large number of streams. If all popular edge regions happened to talk to the same set reflector, they could overload that machine or set of machines, even though the set reflector subsystem as a whole had plenty of spare capacity. Our solution to this problem was to group streams in buckets that we termed *portsets*. We could then assign different portsets to different set reflectors, ensuring that the load on any one set reflector machine did not exceed the machine's capabilities. The details of the solution are described in Section III-B.

### A. Reflector Subscription System

In order to implement subscriptions, we had to start by modifying the commercial streaming servers to propagate end-user requests to the transport layer of our CDN. Since all commercial servers support plug-in extensions, we simply created plug-ins that trap stream requests from end users and propagate them to our transport layer. However, instead of having each streaming server propagate stream subscriptions independently, we have set up our plug-ins to multicast all stream requests on their region's private network. Those requests are then picked up by a separate, in-region process called an edge reflector. Edge reflectors keep track of which servers are requesting a stream, aggregate the requests, and propagate the aggregated requests upstream to the set reflector machines. While all edge reflectors in a region accept subscription requests, only a small subset of them (in the simplest case, one edge reflector) take any action on behalf of those requests. This small subset is selected via a leader election process and is often referred to as the set of *lead edge reflectors*.

When edge reflectors from a region propagate stream requests to set reflectors, they decide which set reflector machine to propagate to by looking up a domain name uniquely associated with the region where the requests originate. For example, the number one leader edge reflector in region 1283 would look up the name n1.r1283.reflector.net to find its parent set reflector. The use of the domain name system

(DNS) as the mechanism for determining the parent set reflector of a region provides tremendous flexibility. It allows us to remap an edge region to a different parent set reflector when connectivity between them deteriorates, or when the parent machine suffers a hardware failure, without having to manage configuration files on individual machines. Furthermore, we get all the scalability and reliability properties of DNS for free.

Once a subscription request for a certain stream has been received by a set reflector, it still needs to be propagated to the appropriate entrypoint machine where the stream enters the CDN. This is slightly more complicated, since set reflectors do not have *a priori* knowledge of a stream's origin. The way this knowledge is built is by requiring entrypoints to *announce* to set reflectors which streams are active on a particular entrypoint. Set reflectors can then use this announced information to chain a subscription to the appropriate entrypoint.

Our experimentation with the system shows that the cost of the subscription and announcement metadata communication is negligible when compared to the volume of the actual stream data being transmitted. Furthermore, the latency associated with propagating the subscription is on the order of tens of milliseconds and is completely overshadowed by the normal startup times associated with streaming players.

### B. Multiple Virtual Reflector Networks: Portsets

Subscriptions and announcements ensure that only watched streams get propagated to edge regions. However, they do not adequately control the path that those streams take through the reflector network. Edge regions simply look up their parent set reflectors and request streams through them. The downside of this approach is that it is hard to bound the number of streams that can be requested through an individual set reflector. Set reflector traffic scales proportionally to two parameters; the number of unique streams it is serving and the number of edge regions to which it is serving them.

Limiting the number of edge regions served per set reflector is relatively straightforward. We can organize set reflectors in set reflector regions which behave similarly to edge regions. A set reflector receiving a request from an edge region simply multicasts that request on its own region's private back end. A lead set reflector aggregates all requests from the region and propagates them to the appropriate entrypoints. Incoming data to the lead set reflector from the entrypoints is also multicasted on the private back end, where it is picked up by nonleader set reflectors and then relayed to the edge regions. This way we can scale the number of edge regions we can serve by increasing the size of a set reflector region. However, managing how many unique streams a set reflector region can handle remains to be addressed.

Our mechanism for managing the number of unique streams per set reflector region is a stream bucketing concept called a *portset*. A portset is no more than a collection of streams that is supposed to be transported through the same set reflectors. We have achieved this by modifying the DNS names that edge reflectors use, to look up their parent set reflectors. Those DNS names contain an additional component that is the portset identifier. For example, a lead edge reflector in region 1283, trying to get a stream in portset 4096, would look up the name n1.r1283.s4096.reflector.net. Once again, the flexibility of the DNS system has proven invaluable for our system. If traffic on particular portsets is low, then the DNS names for those portsets can resolve to the same set reflector. As traffic grows, then the DNS names can be changed to resolve to different set reflectors for different portsets allowing us to scale up or down the reflector network based on customer demand, rather than having to size it based on peak demand.

## IV. Quality Enhancements

Having a distributed scalable CDN is only useful if data can be transported to the edge servers reliably and with high quality. Live streaming is a particularly vexing problem, since there is little margin for error when sending a stream from an encoder to the edge servers. Edge servers typically keep a buffer of a few seconds' worth of stream data. Data arriving at the server gets appended to the end of the buffer while data at the front of the buffer gets streamed to end users. The window of opportunity during which one can recover lost packets is determined by the size of this buffer and is, therefore, only a few seconds wide.

We use two different techniques to ensure relatively loss-free delivery of data to edge servers while incurring minimal additional overhead/cost. The first technique seeks to make each link in our reflector system as reliable as possible, while the second technique attempts to synthesize data across multiple paths to stitch together a complete, lossless stream. To further improve stream quality, we use a third technique called *prebursting*, which attempts to address stream startup issues that arise in a distributed streaming network in which the streaming server only get a stream feed in response to end-user requests.

### A. Packet Loss Recovery

When an end user requests a live stream from an edge server, that server has to in turn get the stream from the encoder that is producing it. Chaining the subscription requests is described in Section III-A, but after a request has been propagated to the stream's entrypoint and data is flowing to the edge server, we need to make sure that the data flows with as little packet loss as possible. One possibility would be to use the Transmission Control Protocol (TCP) as the transport layer between the entrypoints, set reflectors, and edge reflectors, but TCP's recovery mechanism from packet loss has undesirable properties like backoff and bandwidth throttling than can harm an end user's experience. We, therefore, decided to explore recovery schemes on top of a User Datagram Protocol (UDP) transport layer.

While a number of researchers have looked into packet-loss recovery techniques, our goal was to look at actual traces

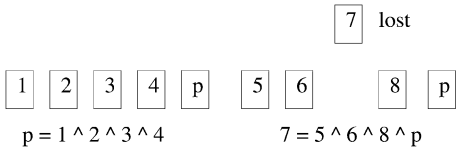


**Fig. 3.** Parity scheme across a single stream.

collected from our early streaming network and focus on recovery techniques that introduced minimal overhead. We collected traces of packet flows that were at least 1 h long (some are longer), and span a million packets or more each. We looked at four different mechanisms for lost packet recovery: parity packets within a stream, parity packets across a combination of streams, a variation of Reed–Solomon codes [9], and plain retransmits.

Fig. 3 shows the idea behind a stream parity scheme. For every $k$ packets of the stream, the entrypoint in our system would introduce a parity packet that is computed as the XOR of the previous $k$ packets. Since packets are not of fixed size, we can assume that smaller packets are padded to the maximum packet size with zeroes. Recovering a lost packet within a parity window is as simple as taking the XOR of the $k$ packets that have arrived at the edge. Clearly, loss of two or more packets within a parity window implies that recovery is no longer possible. A tuning parameter for this scheme is the frequency at which parity packets are inserted into the stream. We evaluated four different variations that incurred overhead of 3.125% (one parity packet every 32 regular packets), 6.25% (one parity packet every 16 regular packets), 12.5% (one parity packet every eight regular packets), and 25% (one parity packet every four regular packets).

Fig. 4 shows the percentage of lost packets that is recovered as a function of the number of parity packets introduced into the packet flow for four different stream traces. Surprisingly enough, the substantially higher overhead schemes do not provide much of an additional benefit. The reason for this is that lossiness tends to be bursty rather than random. Therefore, even high-overhead schemes are unlikely to be able to recover when a number of contiguous packets are lost.

The second scheme we evaluated was parity packets across multiple streams. The idea is depicted graphically in Fig. 5. Instead of thinking of packets as a one-dimensional structure, we organize them logically in two dimensions. Parity packets then take the form of a row inserted along the second dimension. For example, in the figure, we show packets organized in rows of eight packets and a parity row inserted for every three data rows. This allows us to tolerate bursty losses of contiguous data packets, since the parity packets are introduced along the second (noncontiguous) dimension. The disadvantage of this approach is that parity packets are relatively far away from the data packets that they complement; thus, recovery of a packet can take longer and may not happen in time for the recovered packet to be useful. However, this problem can be mitigated by implementing this parity scheme across packets from multiple streams rather than a single stream. Combining packets from

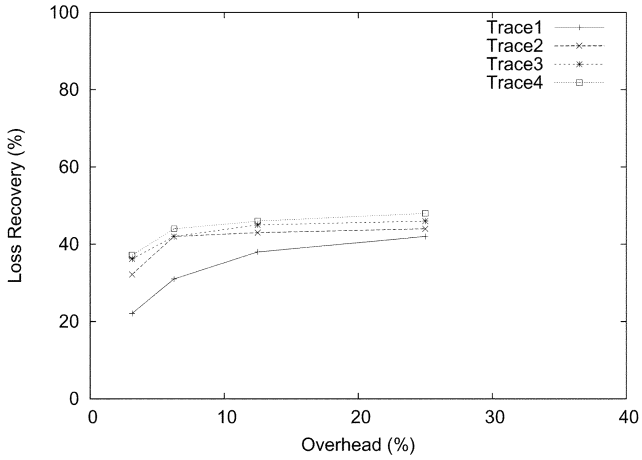


**Fig. 4.** Overhead and recovery rates of parity scheme across a single stream.

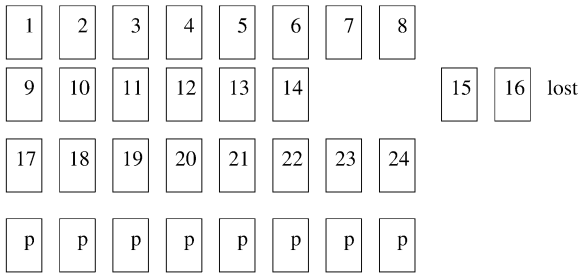


**Fig. 5.** Parity scheme across multiple streams.

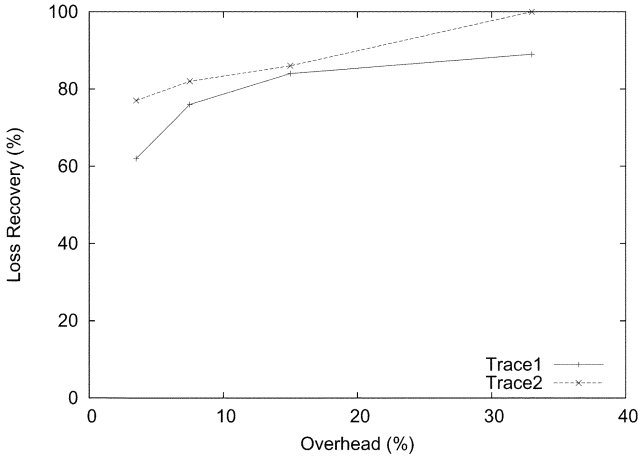


**Fig. 6.** Overhead and recovery rates of parity scheme across multiple streams.

multiple streams implies a higher packet rate and, thus, less wait time before a lost packet can be recovered. This way, recovery can happen within the time frame allowed for any single stream while maintaining the property that bursty packet loss can be tolerated.

Fig. 6 shows the efficacy of this approach in recovering lost packets for two different multiple-stream traces that we have collected. As can be seen, it is almost twice as successful than the simple parity scheme in recovering packets for comparable amounts of overhead.

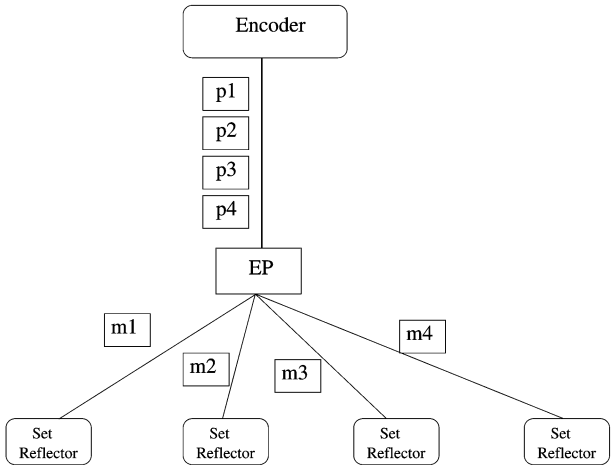


**Fig. 7.** Multipath error correction scheme.

The third scheme we evaluated took advantage of the fact that our system allows for delivery of packets across multiple paths. Rather than trying to introduce redundant packets within a single link, we rely on a variation of Reed–Solomon codes to compute $m$ derivative packets from $k$ original packets. The new derivative packets are derived from the original $k$ packets through simple linear functions and have the property that the presence of at least $k$ out of the $m$ derivative packets is sufficient to compute the $k$ original packets by solving a simple system of linear equations. The scheme is shown graphically in Fig. 7. We evaluated two different versions of this scheme where we compute three (50% overhead) and four (100% overhead) derivative packets for every two original packets. Schemes with larger $k$ and $m$ value result in more complex equations and can end up being too computationally intensive for edge servers to be successful. As we can see in Table 1, taking advantage of multipath delivery and Reed–Solomon encoding is quite effective in recovering lost packets, with recovery percentages ranging between 91.9% and 99.5%. Nonetheless, the fact that this scheme requires at least 50% overhead makes it an unlikely candidate for packet recovery.

The last scheme we evaluated was a best effort retransmit system. We modified our transport system to keep a buffer of the most recent $n$ packets it had forwarded to each destination. Destinations examine the sequence numbers of received packets and decide that a packet with sequence number $i$ is lost if they receive packet $i + k$ without having received packet $i$. They then simply request the lost packet from their upstream parent, which will send it to them if it exists in its buffer of recently sent packets. Notice that this is a best effort approach; a child node will not wait for a missing packet indefinitely, and parent nodes only promise that they will hold packets for retransmits for a short period. An important tuning parameter for this scheme is the out-of-order tolerance $k$ that receiving nodes need to have. Clearly, packet reordering is a common phenomenon on the Internet, and we need to ensure that we do not request and resend packets simply because they have arrived out of order. On the other hand, having a very large out-of-order window could result in

**Table 1**
Overhead and Recovery Rates of Multipath Error Correction Scheme

| TraceId | Overhead | Loss Recovery |
|---|---|---|
| r1003-0411 | 50% | 93.9% |
| | 100% | 98.9% |
| r419-0331 | 50% | 89.2% |
| | 100% | 91.9% |
| r421-0322 | 50% | 96.2% |
| | 100% | 98.7% |
| r686-0322 | 50% | 98.7% |
| | 100% | 99.5% |

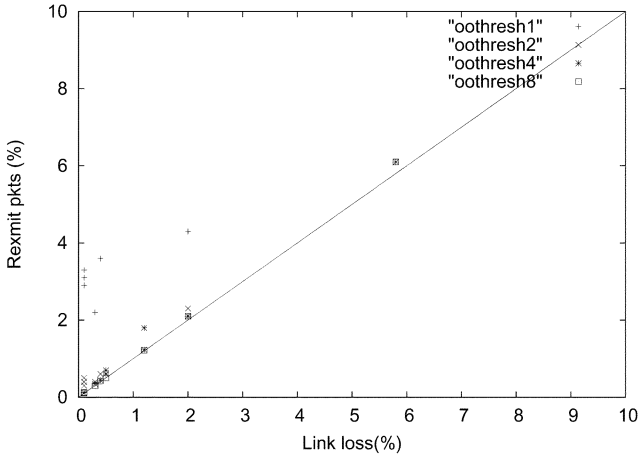


**Fig. 8.** Retransmit overhead as function of packet loss for different out-of-order tolerance thresholds.

lost packets being requested and resent too late to be useful to the streaming server. The retransmit scheme is extremely effective in recovering lost packets. More than 99% of all lost packets are recovered using this scheme. Fig. 8 shows the overhead incurred by the retransmit scheme for four different values of the out-of-order window. As can be seen from the figure, a window that only tolerates simple packet transpositions (i.e., packet $n+1$ arrives before packet $n$) can occasionally result in relatively high retransmission costs. Fortunately the overhead diminishes quickly for window sizes of two or more; thus, we can have both quick loss detection and low retransmit overhead.

Given the evaluation results presented, we decided in the end to go with a retransmit scheme in our CDN. It provided good loss recovery properties (as good as or better than the other three schemes), and had the additional advantage that it induced overhead only when packets were indeed lost, rather than all the time.

### *B. Adaptive Multipath Transmission*

Even with retransmits, it is possible for a particular link in our transport layer to remain lossy or, even worse, to be down. In order to address this failure case, we took advantage of the fact that our system can easily support multipath transmission. Packets originating from a particular entrypoint can be replicated and sent across multiple set reflectors to a single edge region. The packets are then recombined at the edge region, duplicates are removed, and the clean, lossless stream is handed to the streaming server that

can then send it to end users. While multipath transmission can provide extremely high quality and virtually loss-free transmission of live streams, it has one great disadvantage: it is exceedingly costly, especially for unpopular streams. Imagine a case where a single user requests a live stream from an edge server. That stream has to leave the entrypoint, arrive at a number of set reflectors equal to the number of paths that we want to use, and then be forwarded to the edge region so it can be recombined before being streamed to the end user. If we assume three transmission paths to ensure lossless delivery, then six copies of the stream have to travel through the reflector network for the single copy served to the end user. Unfortunately, the economics of a such a scheme are unacceptable, especially since the additional paths are needed only rarely.

We have modified the subscription system described in Section III-A to allow us to use multiple path transmission only when it is necessary. The modifications were relatively straightforward, with one notable exception described below. We started by modifying the streaming servers to provide stream quality information to the edge reflectors. Therefore, it is possible for the streaming servers to tell the reflector system whether they are getting a good quality stream or not. The edge reflectors can then adapt the number of paths used in fetching the stream to a region based on this quality information. As described in Section III-A, edge reflectors elect a small number of leaders responsible for fetching streams into the region. We can change the number of leaders to be elected (and, thus, the number of copies of a stream fetched into the region) based on the quality information provided by the streaming servers. A long period of good quality information triggers a reduction in the number of leaders and, consequently, the number of paths in use. A short period of bad quality information triggers an increase in number of leaders and consequently the number of paths in use. Clearly, the lowest possible number of paths in use is one, while we have capped the maximum number of paths that can be in use to three.

A complication to this system arises when the edge region has determined that a single path provides sufficient data quality but the path through which the data is arriving suddenly goes down.[1] In that case, we need to rapidly detect the loss of the one active link and recover by bringing into use the secondary and/or tertiary paths for that edge region. Detection has to happen quickly enough to ensure that end-user experience is not affected.

In order to achieve this kind of rapid detection, we have modified our entrypoint nodes to ensure that each stream has at least 1 packet/s. For requested streams that fail to deliver a packet for 2 s in a row and are using a single path, the edge reflector processes assume that the primary link is dead and initiate subscription requests through the secondary and tertiary links in order to recover. Experience running the adaptive multipath transmission system in conjunction with retransmits indicates that it results in close to lossless streaming at relatively low cost. Packet loss for streams reconstructed at edge regions is almost always below 0.1%, while the average number of fetch paths used per stream varies between 1.1 and 1.2 across the network. The fact that the average number of paths is greater than one indicates that some kind of multipath transmission is necessary in order to achieve high quality under all circumstances, but it is important to make it adaptive in order to keep costs under control.

[1]A path can go down either because the upstream node has died, because the network link between the upstream node and the edge region has gone down, or because the downstream node within the edge region has died.

### C. Prebursting

The final technique used in our transport layer in order to improve quality is prebursting. The idea behind prebursting is to deliver a stream to a server at a higher rate than the encoded rate for the first few seconds of the streaming session's life. We conducted studies showing that such a delivery behavior results in better end-user experience mostly through reduced buffering time. The reason that prebursting reduces rebuffering and initial buffering times is that it allows the streaming server to build its own buffer of the stream rather quickly. It is, thus, able to tolerate small fluctuations in the delivery of packets by the reflector network and avoid buffer underflows (a situation where the server needs to send the next packet to the end user but has not received it from our transport layer). Note that the latest Windows Media Server (WMS) [6] uses a similar technique in the communications path between the server and end users.

Clearly, it is not possible to deliver data at a higher rate than the encoder is producing it, unless we save some data that the encoder has produced in the past and we start delivery with this previously produced data. This is exactly the approach we have taken. We take advantage of the fact that we already keep data around for retransmit purposes and reuse the retransmit buffers as the prebursting buffers. When a request first arrives at a set reflector or entrypoint for a particular stream, instead of sending the most recently received packet, we send the oldest received packet for that stream that is still in the retransmit buffer. We then continue to send packets out of the retransmit buffer at a rate that is a multiple of the rate at which packets arrive from the encoder. Once all packets in the retransmit buffer have been sent, we simply forward packets as they arrive from the encoder.

One nice side effect of the prebursting approach is that it makes recovery from failed paths seamless. When the adaptive multipath transmission system notices that a link has gone down and starts a new subscription request through a secondary or tertiary path, it expects to receive a preburst from those paths. Given that the preburst starts with a packet from some time in the past rather than the most recent packet produced by the encoder, packets that were missed during the fault detection interval will be received as a part of the preburst and assembled into a lossless stream that can be delivered to the streaming server. Fig. 9 shows the amount of rebuffering experienced by end users both in the presence and in the absence of prebursting. As can be seen, prebursting is instrumental in reducing rebuffering effects and greatly increases the quality of the end-user experience. We have also seen comparable improvements in the stream startup times experienced by end users. Overall, prebursting allowed us to

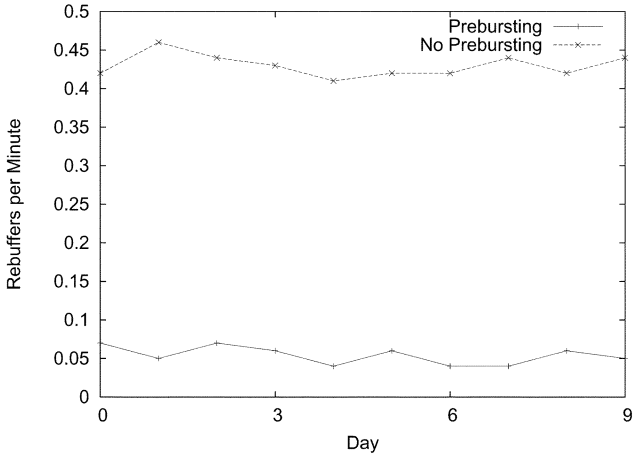


**Fig. 9.** Effect prebursting on stream rebuffering.

improve end-user quality with only a modest engineering effort, since we were able to leverage the same mechanisms used in lost packet recovery.

## V. Reliability Enhancements

The system that we have presented so far is fault tolerant in almost all of its aspects. Edge regions are interchangeable, and the mapping and load balancing system ensures that only live edge regions with good connectivity are being used to serve end users. The reflector system has three-way redundancy and can be reconfigured to exclude dead machines with simple DNS changes. The only parts of the system that are vulnerable to machine and network failures are the entrypoints. Since an encoder connects to a particular entrypoint to provide data, it is possible for that entrypoint to die or lose connectivity to the encoder.[2] To recover from this type of failure, we have modified our system to tolerate entrypoint faults. The solution we have adopted is described in the following section.

### A. Fault-Tolerant Entrypoints

Our fault-tolerant entrypoint solution relies on a distributed leader election algorithm. The main difference between typical leader election algorithms and our approach is that in our case the candidates for leader do not actually communicate with each other. Rather, they rely on outside observers (the set reflectors) to provide the voting information as to who should be the leader. This has the benefit of leveraging existing communication paths between entrypoints and set reflectors, while supplying redundant paths for entrypoints to obtain leadership information about each other. Fig. 10 depicts the data flow between set reflectors and entrypoints.

The system starts by distributing a configuration file to every entrypoint and set reflector that contains an ordered list of candidate entrypoints for every stream in the system. The first entry in that list is the default entrypoint pulling the stream from the encoder, while the secondary and tertiary entries are the fallback choices, should the primary choice fail.

Entrypoints provide a heartbeat packet each second and periodically announce which streams they can pull from encoders to the set reflectors. The set reflectors themselves ensure that the lead entrypoint for every stream in the configuration file is both alive and can actually can pull the streams for which it is leader.

Failures fall in two categories. The first category is the death of the entrypoint machine itself. When an entrypoint machine dies, its heartbeat ceases to arrive at set reflectors. The set reflectors then produce a list of the streams for which the dead entrypoint was a leader and publish it to the secondary and tertiary entrypoints for those streams. The secondary and tertiary entrypoints for those streams notice the information published by the set reflectors and attempt to pull the orphaned streams from their encoders. In order to ensure that only one of the backup entrypoints ends up pulling the orphaned streams, there is a delay built into the system for the tertiary entrypoint. If the secondary entrypoint succeeds in pulling the stream, it will start announcing its existence to the set reflectors. They in turn will modify the information that they publish to indicate that the primary entrypoint is dead and that the secondary entrypoint has successfully assumed its duties. The tertiary entrypoint will notice this change in information and will cease any efforts to pull the stream.

The second fault category is the loss of connectivity between the primary entrypoint and the encoder. Detection of this failure happens when the primary entrypoint stops announcing the existence of the stream originating at the unreachable encoder. The set reflectors notice the absence of the announcement and publish the relevant information. Failures of this type are detected less rapidly than entrypoint deaths, because the mechanism relies on announcements which are sent less frequently than the heartbeat packets. However, in all other respects, the mechanism for dealing with this failure is identical to the one described above, with the secondary and tertiary entrypoint leader attempting to assume the duties of the primary leader with respect to the problematic stream. If either the secondary or the tertiary entrypoint have connectivity to the encoder, the stream will resume shortly after the fault is noticed.

Returning to regular operational mode once a fault has been repaired is also straightforward. When the primary entrypoint comes back to life or can reconnect to an encoder, it starts providing the necessary heartbeat and announcement information to the set reflectors. They then stop publishing the information, indicating that action needs to be taken by the backup entrypoints. The absence of this information causes the secondary and tertiary entrypoints to stop pulling the stream from the encoder, and the primary leader resumes its regular role.

It is possible for the system to transiently be in states where more than one entrypoint is connected to an encoder and pulling a stream. Set reflectors (and edge reflectors) have knowledge of the relative ordering of entrypoints with respect to each stream. When they see packets for a particular

[2]It is also possible for the encoder to die, but since this is a machine not in our control, there is little we can do to recover from this kind of failure.

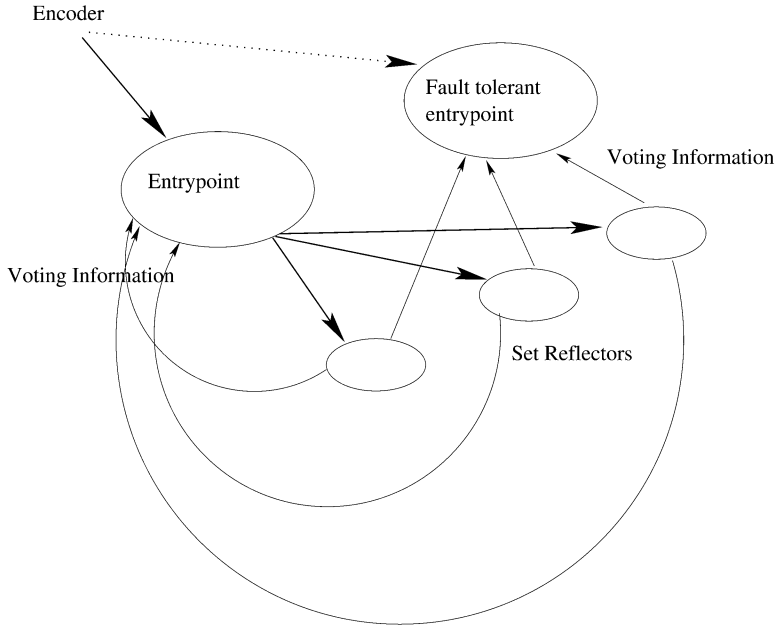


**Fig. 10.** Fault-tolerant entrypoint diagram.

stream originating from multiple entrypoints within a short configurable period, they will select the one with the highest leader number and ignore packets from the lower leader numbers.

We have tested the behavior of the system by inducing both types of faults and checking the impact on an end users watching a stream. Death of an entrypoint is almost unnoticeable from an end user's perspective. Most times streams continue to play with no problems, while occasionally a rebuffering event may be noticed by the end user. The main reason for this benign behavior is that heartbeats occur every 1 s; thus, death of an entrypoint is noticed and recovered quickly enough to prevent the streaming server's and/or client's stream buffer from draining. Loss of connectivity between an entrypoint and an encoder takes a little longer to be noticed and often results in a rebuffering event or even a disconnect for an end user. However, the stream becomes available again with only a few seconds of downtime and no manual intervention is necessary.

With the introduction of fault-tolerant entrypoints, we have eliminated the last single point of failure of our system. Even when machines and network connections fail, streams will continue to be available for end users who are outside the failed networks. Furthermore, in most cases, existing users in nonfailed networks will remain unaware that a fault even occurred. This solution is extremely attractive from a customer's perspective, since it ensure reliable stream delivery with no modifications to their encoding infrastructure and guarantees automatic handling of faults without the need for human intervention.

## VI. Performance Measurements

While there is no standard for performance measurements with respect to streaming, a number of streaming session attributes can be shown to be important to an end user's experience. Akamai has developed a proprietary agent technology to measure streaming performance and has deployed a network of over 50 agents to take performance measurements. All measurements discussed in this section come from approximately 30 of these agents, testing streams delivered by the Akamai network hourly over the course of a month, representing 41 391 data points. More detail on our performance methodology can be found in [10].

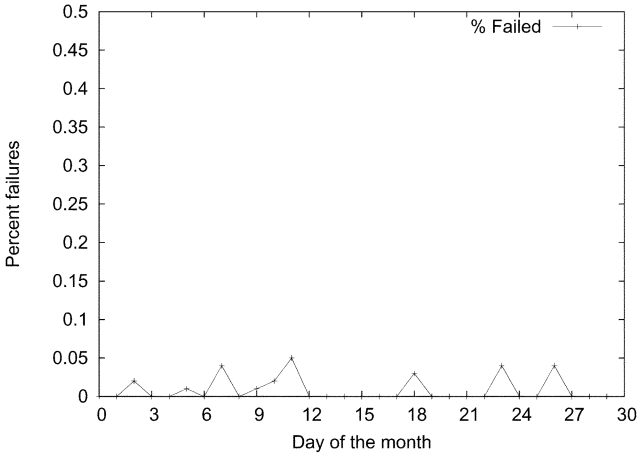


**Fig. 11.** Failure rate over a period of a month.

Our first metric is failure rate—how often an end user's attempt to play a stream is successful or not. Fig. 11 shows the failure rate of our system over a period of a month. We have aimed for a failure rate of less than 0.1%, and the current architecture meets or exceeds this goal comfortably. A second important metric is number of rebuffers per minute. This metric captures how often an end user's experience is interrupted by a frozen stream while the player waits for data to arrive from the server. Fig. 12 shows the rebuffer rate of our system over a period of a month. We aimed for a rebuffer

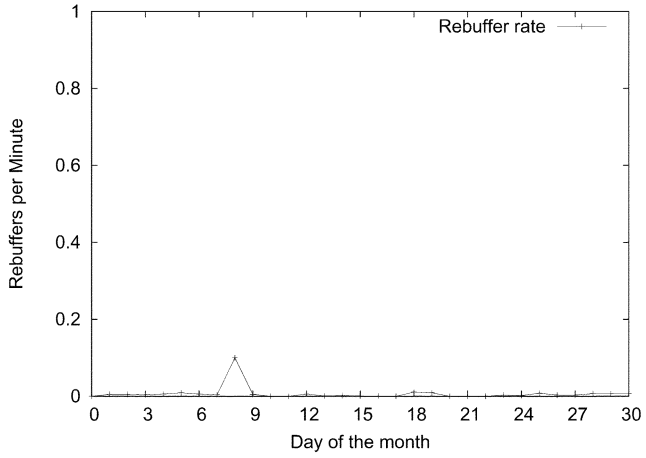


**Fig. 12.** Rebuffer rate over a period of a month.

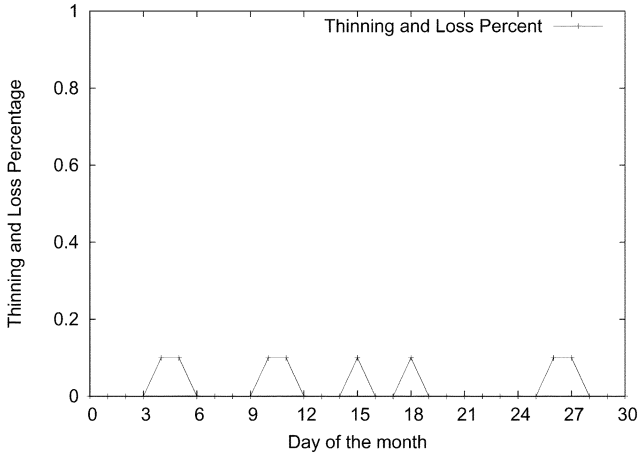


**Fig. 13.** Thinning/loss percentage over a period of a month.

rate of less than one rebuffer per hour. Once again, the current architecture comfortably meets or exceeds this goal.

We have also chosen to measure one additional attribute of streaming sessions: thinning/loss as a percent of ideal bandwidth. Streams are typically generated by an encoder at a bandwidth rate determined by the stream's producer. However, servers do not have to deliver streams to end users at the rate at which they were encoded. A server that is overload or has poor connectivity to the client can choose to "thin" a stream by not sending certain frames, thus putting less stress on both the server and the link that connects it to its clients. This effect does not manifest itself as packet loss, since all packets sent by the server are received by the client. Thinning results in suboptimal user experience in ways similar to those caused by regular packet loss. By looking at the rate of the stream as received by the client versus the rate at which the stream was encoded, we can combine both effect into a single metric that is easier to measure and is more informative than either metric in isolation. Fig. 13 shows how our system performs with respect to the thinning/loss metric. As can be seen, thinning/loss is consistently less than 0.1% of the stream's intended encoding rate.

## VII. Related Work

CDNs have received substantial attention from both industry and academia. A number of commercial systems are operational (including our own), and many academic and other research institutions have studied them in some detail, both in the context of static Web content [3], [4] and streaming content [1], [2].

In the context of streaming CDNs, a number of studies have sought to address the issue of effective media file caching. SOCCER [12], Mocha [13], and Middleman [11] all address the problem of effective media caching on streaming servers. These systems use segmentation techniques, prefetching of data, and cooperative caching among multiple servers to improve media server caching performance. Van der Merwe *et al.* [14] and Cherkasova and Gupta [15] also present characterizations of streaming video traffic and show that various parts of a clip have different probabilities of being viewed. Thus, they conclude that content segmentation and caching of selective segments is more cost effective and offers better performance than caching of whole media files. While our system deals with video-on-demand content in ways similar to those described by these studies, the focus of this paper is live streams that obviously do not lend themselves to caching.

Yajnik *et al.* [16] present a study of packet loss characteristics for streaming media and show the temporal correlation between loss. Veloso *et al.* show similar findings in [17]. Our work verifies their finding on the burstiness and autocorrelation of packet loss. However, we also present a study on a number of correction schemes and show that the simplest scheme for packet recovery (retransmits) has the best properties when both overhead and recovery percentage are taken into account. Using multiple paths for media delivery is also a concept that has been studied in detail [18]–[21]. However, most of those systems make the assumption that they can control the coding scheme for the content that they transport. In our case, encoders are part of commercial offerings and we have little control over them. Therefore, we cannot prioritize between packets being delivered and must deliver all packets produced by an encoder at reasonable transport costs. Our experience indicates that adaptive redundant paths coupled with a retransmit scheme provide the desired functionality at a very reasonable cost.

## VIII. Conclusion

In this paper, we have discussed the design decisions we have made while building a CDN for live streaming. We have described how to achieve high degrees of scalability, quality, and reliability by focusing on modular design and eliminating single points of failure. We have evaluated multiple techniques for delivering data to edge servers before deciding on a combination of retransmits and multiple paths as our approach of choice. Furthermore, we have shown that delivery of stream data at rates higher than the encoded rate for the first few seconds of a session can significantly improve an end user's quality. Finally we have introduced a mechanism for eliminating single points of failure at the entrypoints of

our system. The described system currently serves millions of streams per day to end users across the world, and has scaled to 80 000-plus concurrent users and 16 Gb/s of traffic.

Despite the success of the developed system, a number of issues remain as interesting technical questions. We would like to determine whether the reflector hierarchy itself can be bypassed altogether for unpopular streams and how the system would have to be modified to handle the transition of a stream from the unpopular to the popular category and *vice versa*. It would also be interesting to have edge regions choose their parent set reflectors in a completely dynamic fashion and not have to rely on bucketing techniques for load balancing. The multipath transmission system can potentially benefit from modifications that would allow it to pick the best path among its choices, rather than the number of paths necessary to provide good quality. Finally, the fault-tolerant system can be further tuned to ensure that fault recovery transitions go completely unnoticeable by end users. Those questions notwithstanding, the existing system offers tremendous benefits over both centralized and naive distributed CDN implementations, and we believe it is a good compromise between engineering and operations cost and customer benefit.

## REFERENCES


[1] C. Cranor, M. Green, C. Kalmanek, D. Shur, S. Sibal, and J. V. der Merwe, "Enhanced streaming services in a content distribution network," *IEEE Internet Comput.*, vol. 5, pp. 66–75, July/Aug. 2001.

[2] S. Wee, W. Tan, J. Apostolopoulos, and S. Roy, "System design and architecture of a mobile streaming media content delivery network (MSD-CDN)," Streaming Media Syst. Group, HP Labs., Palo Alto, CA, Tech. Rep. HPL-2003-77, 2003.

[3] I. Lazar and W. Terrill, "Exploring content delivering networking," *IT Professional*, vol. 3, pp. 47–49, July/Aug. 2001.

[4] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek, "The measured performance of content distribution networks," *IEEE Internet Comput.*, vol. 5, pp. 66–75, July/Aug. 2001.

[5] Akamai [Online]. Available: http://www.akamai.com

[6] An introduction to Windows media services 9 series (2003). [Online]. Available: http://download.microsoft.com/download/winmediatech40/wp/1a/W9X2KMeXP/EN-US/wms9sintro.exe

[7] Helix DNA server architecture overview (2002). [Online]. Available: https://helix-server.helixcommunity.org/2003/devdocs/architecture.html

[8] Quicktime streaming server administration guide (2002). [Online]. Available: http://a336.g.akamai.net/7/336/51/f38d6845c55f90/www.apple.com/server/pdfs/QT_Streaming_Server.pdf

[9] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, June 1960.

[10] Akamai streaming: When performance matters (2002). [Online]. Available: http://www.akamai.com/en/resources/pdf/whitepapers/Akamai_Streaming_Performance_Whitepaper.pdf

[11] S. Acharya and B. Smith, "Middleman: A video caching proxy server," presented at the Workshop Network and Operating System Support for Digital Audio and Video, Chapel Hill, NC, 2000.

[12] E. Bommaiah, K. Guo, M. Hofmann, and S. Paul, "Design and implementation of a caching system for streaming media over the Internet," in *Proc. 6th IEEE Real Time Technology and Applications Symp.*, 2000, pp. 11–121.

[13] R. Rejaie and J. Kangasharju, "Mocha: A quality adaptive multimedia proxy cache for Internet streaming," presented at the Workshop Network and Operating System Support for Digital Audio and Video, Port Jefferson, NY, 2001.

[14] J. van der Merwe, S. Sen, and C. Kalmanek, "Streaming video traffic: Characterization and network impact," presented at the Workshop Web Content Caching and Distribution, Boulder, CO, 2002.

[15] L. Cherkasova and M. Gupta, "Characterizing locality, evolution, and life span of accesses in enterprise media server workload," presented at the Workshop Network and Operating System Support for Digital Audio and Video, Miami Beach, FL, 2002.

[16] M. Yajnik, S. Moon, J. Kurose, and D. Towsley, "Measurement and modeling of the temporal dependence in packet loss," in *Proc. INFOCOM*, vol. 1, 1999, pp. 345–352.

[17] E. Veloso, V. Almeida, W. Meira, A. Bestavros, and S. Jin, "A hierarchical characterization of a live streaming media workload," Boston Univ., Boston, MA, Tech. Rep. 2002-014, 2002.

[18] J. G. Apostolopoulos, "Reliable video communication over lossy packet networks using multiple state encoding and path diversity," in *Proc. SPIE Conf. Visual Communications and Image Processing*, vol. 4310, 2001, pp. 392–409.

[19] N. Gogate, D.-M. Chung, S. S. Panwar, and Y. Wang, "Supporting images and video applications in a multihop radio environment using path diversity and multiple description coding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 12, pp. 777–792, Sept. 2002.

[20] T. Nguyen and A. Zakhor, "Path diversity with forward error correction (PDF) system for packet switched networks," in *Proc. INFOCOM*, vol. 1, 2003, pp. 663–672.

[21] J. Chakareski and B. Girod, "Rate-distortion optimized packet scheduling and routing for media streaming with path diversity," in *Proc. IEEE Data Compression Conf*, 2003, pp. 203–212.



**Leonidas Kontothanassis**, photograph and biography not available at the time of publication.

**Ramesh Sitaraman**, photograph and biography not available at the time of publication.

**Joel Wein**, photograph and biography not available at the time of publication.

**Duke Hong**, photograph and biography not available at the time of publication.

**Robert Kleinberg**, photograph and biography not available at the time of publication.

**Brian Mancuso**, photograph and biography not available at the time of publication.

**David Shaw**, photograph and biography not available at the time of publication.

**Daniel Stodolsky**, photograph and biography not available at the time of publication.