First publ. as: Washington University Computer Science Technical Report WUCS-01-02, March 2001

Fuzzycast: Media Broadcasting for Multiple Asynchronous Receivers

Marcel Waldvogel, Wei Deng, Ramaprabhu Janakiraman
Department of Computer Science
Washington University in St. Louis
St. Louis, MO 63130-4899
{mwa, wdeng, rama}@arl.wustl.edu

Technical Report WUCS-01-02

Abstract

When using an on-demand media streaming system on top of a network with Multicast support, it is sometimes more efficient to use broadcast to distribute popular content, especially when client demand is high. There has been a lot of research in broadcasting on-demand content to multiple, asynchronous receivers. In this paper, we propose a family of novel, practical techniques for broadcasting on-demand media, which achieve lowest known server/network bandwidth usage and I/O efficient client buffer management, while retain the simplicity of a frame-based single channel scheme. We also propose playout scheduling strategies that make it practicable for serving both constant bitrate (CBR) and variable bitrate (VBR) media.

1 Introduction

Recent advances in technology have made on-demand streaming of digital media feasible. However, for several years these systems have not achieved the notable commercial success that technology seemed to promise. The primary reason is that traditional on-demand systems do not scale well. All available server bandwidth tends to get quickly swamped by client demand, forcing providers to invest in expensive network and hardware resources to ensure quality of service.

The three principal metrics for measuring the performance of on-demand streaming systems are

Bandwidth: This could be network, server or client bandwidth.

Delay: While on-demand media in its truest sense is instantaneous, most discussions on broadcast-based on-demand streaming systems allow a reasonable time interval between request and playout.

Buffer Space: In an on-demand streaming system in which segments can arrive out of order, the client requires memory or persistent storage to store out-of-order segments. In some broadcast schemes, peak requirements can run to several megabytes.

The most popular techniques rely on efficient broadcast mechanisms to simultaneously satisfy the demands of multiple clients without suffering a linear increase in bandwidth usage. Prior studies of request patterns in video rentals [DSS94] suggest that a large fraction of these requests (40–60%) are for a small number (10–20) of videos. This statistic suggests that using some form of broadcast for the most popular media can be very effective in scalable on-demand media distribution.

We present here a family of highly efficient schemes for lowest server and network bandwidth that are known, as well as efficient client resource usage. They are also arguably the simplest solutions that are available. In the next few sections, we discuss these schemes both in theoretical basis and in practical implementation, performance comparisons with the best possible and the best extant schemes are provided for different issues. In Section 2 we will discuss a new approach to address the optimal server and network usages, as well as optimal average bandwidths at clients. Both constant bitrate and variable bitrate media will be discussed. In Section 3 we present two heuristic schemes for efficient client buffer management.

1.1 Background

The techniques we describe apply to an on-demand streaming system, which comprises a central server that distributes digital media to clients over a distribution network. We assume that the network supports a broadcast primitive and that broadcasting a piece of data to a group of clients is in some way more efficient than unicasting it to each client. The server stores a set of *movies* from which each client is free to choose. A movie is a piece of media data that consists of a number of fixed-size blocks of data (*frames*) that are always consumed in serial order. For convenience, we assume a frame can be transmitted over the network without further fragmentation, although this is not a prerequisite for our work.

Clients arrive at times of their choosing, request the server for movies and after an optional waiting period w, get out-of-order frames. Client uses local cache which consists of internal and external memory to hold the frames that are not needed to be played out soon. The media player consumes the frames in the cache from beginning to end in serial order. For the rest of this paper, let the unit of time (an *instant*) be that time required to consume a single frame, so that each client spends $n + w$ instants on a movie of n frames. This, of course, assumes that the content is constant bit-rate. In Section 2.5 we mention that this may be extended to include variable bit-rate media. We also assume that clients arrive synchronously with instants and are able to receive and store frames transmitted during the instant of their joining. In practice, this is not an unreasonable assumption since an instant, by our definition, would correspond to roughly 33 milliseconds for 30 frames-per-second video. This sequence is illustrated in Figure 1.

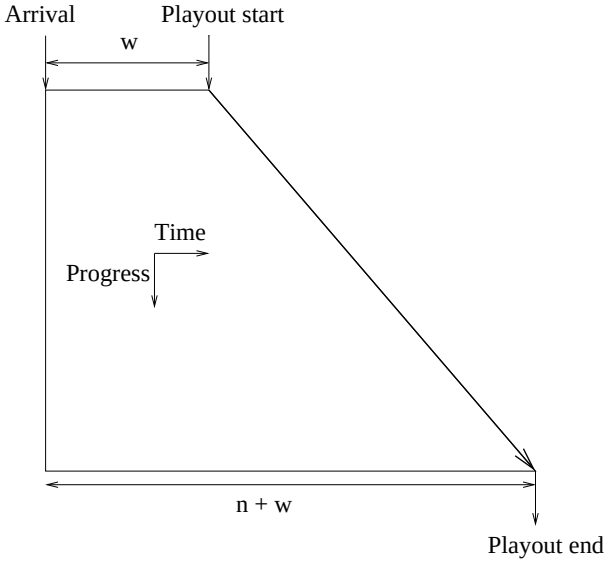


Figure 1: Playout

1.2 Related Work

One of the earliest proposals for bandwidth-efficient on-demand streaming system was *Batching* [DSS94], where the Server aggregated requests that came close together in time. Subsequently, many periodic broadcast methods have been proposed, while some theoretical lower bounds [EVZ99, SGT99] have been obtained.

Pyramid Broadcasting [VI96] divides data into segments of geometrically increasing sizes and broadcasts them on a few high bandwidth streams. A derivative, Skyscraper Broadcasting [HS97] offers much better performance.

More currently, a family of protocols related to Harmonic Broadcasting [JT97, PCL98a] seems to be the most promising insofar as the bandwidth-delay tradeoff is concerned. These protocols split a movie into fixed-size segments and broadcast these segments in streams of harmonically decreasing bandwidth. Segments are groups of adjacent frames and are hence of much coarser granularity than frames.

One variant, Polyharmonic Broadcasting [PCL98b], while in theory offering near-optimal performance, unfortunately encounters formidable obstacles in implementation, since it requires each client to simultaneously subscribe to a large number of low-bandwidth streams. Another related protocol, Pagoda Broadcasting [PCL99] sacrifices some performance for the ability to use a few, constant-bandwidth streams.

For the client side, buffering problem can be formulated as an external memory sorting problem involving random insertions and minimum element deletion. A heap or priority queue based algorithm naturally suggests itself. One high performance variant is a radix-heap based structure, which stores frames in multi-level buckets according to an optimal radix. According to a comparative analysis performed by [BCFM99], radix heaps are the best extant scheme when the keys are integers in a pre-defined range. Another efficient scheme, based on a collection of fixed-size lists, is presented by the same paper. However, when applying these two schemes to our system, they both neglect to take linear access into account, resulting in suboptimal performance. We have used a modified version (which improves the efficiency) of the algorithms presented in [BCFM99] for performance comparison. But the result still shows that our schemes work better.

2 Bandwidth

An important tradeoff in designing the on-demand streaming system is that of server bandwidth versus client delay; most work has been to improve this performance: That is, given one, to minimize the other. In this section, we present an efficient technique to achieve near-optimal performance in server and network/client bandwidth.

2.1 Fuzzycast

2.1.1 Theoretical Basis

Consider the broadcast of a "popular" movie that consists of n frames. Assume the frames are to be broadcasted in a way that satisfies the on-demand requirements of clients with different join-times. Now, a client with a join-time of t and a wait-time of w will require frame f at time t_f no later than time $t+w+f$, i.e., $t \leq t_f \leq t + w + f$. For this to hold true for all t, it is necessary and sufficient that there is at least one transmission of frame f in every stretch of $w + f$ instants. The optimal way to do this is to broadcast frame f every $w + f$ instants. This is the theoretical basis for our work.

```
for (f ← 1; f ≤ n; f ← f + 1) {
    for (t ← f + w; t ≤ t_max; t ← t + f + w) {
        transmit(f, t);
    }
}
```

Figure 2: Algorithm NAIVE

Here the *transmit* function schedules f for transmission during time t in a transmission queue.

The bandwidth usage of algorithm NAIVE (Figure 2) at both server and client ends is: $\sum_{f=1}^{n} \frac{1}{w+f} \approx \log \frac{n+w}{w}$. Earlier work [EVZ99] has determined that this is optimal for any broadcast based scheme.

Although Algorithm NAIVE is simple, elegant and optimal, a fatal flaw renders it unusable. The number of frames scheduled for transmission at time t is the number of integers $i \geq w$ such that i divides t. This function, the factor function, is extremely spiky, varying from 2 for prime t to new highs for highly composite t. It is because of this reason that Algorithm NAIVE has been considered and then regretfully abandoned by various researchers. Two approaches exist to counter this. The stream-based approach, instead of transmitting frame[1] f every $f + w$ instants, transmits it continuously in a seperate channel of bandwidth $\frac{1}{f+w}$ times the base bandwidth. Most of the harmonic broadcasting protocols take this approach, with Polyharmonic broadcasting [PCL98a] offering the best performance. This promises uniform bandwidth access, but also runs into these difficulties:

1. In all forms of Harmonic broadcasting, segment sizes are constrained by the initial delay. To gain user acceptance, the initial delay has to be a small fraction of movie length, requiring a movie to be transmitted over a large number of streams. Managing hundreds of low bandwidth streams over a packet network is a nontrivial exercise. In Polyharmonic broadcasting, the number of streams increases by an additional factor of m, $m = 4$ being a reasonable value. In practical terms, using Polyharmonic Broadcasting for a single 2-hour, 1.5 Mbps movie with a 30 second initial delay translates to maintaining 960 streams with server bandwidths that vary from 1.5 Mbps to a few hundred bps.

2. It only partially solves the problem, because at some level, streams map to network packets. Since packets cannot be arbitrarily small, multiple transmissions of later frames have to be aggregated. Aggregating again brings in the scheduling issues we have just described.

3. Usually, information cannot be retrieved from partial frames and they must be discarded. Unless error correction techniques like Forward Error Correction (FEC [NBT98]) are used, transmitting a segment over an extended period and over multiple packets increases the likelihood of this happening for frames in later segments.

[1] or segment

The alternate approach is to time-division multiplex the frames in a deterministic way that avoids the scheduling conflicts of Algorithm NAIVE. Pagoda broadcasting [PCL99] does this, but sacrifices some performance in the process because it has to settle for suboptimal schedules. We now propose a simple frame-based algorithm that is at least as efficient as the best stream-based algorithm.

2.1.2 The Basic Algorithm

As we have seen, Algorithm NAIVE is infeasible since it results in spiky bandwidth usage. Our solution to this is (Figure 3): Whenever a frame f has to be scheduled at an instant that has used up the bandwidth allotted to frames $1 \cdots f$ ($= \log \frac{f+w}{w}$), we allow it to stray from its scheduled instant to one of its neighbouring instants wth bandwidth to spare. The intent is to spread or smear out a bandwidth peak over time, flattening peaks and filling up troughs, while ensuring that for each frame f, the gap between succesive transmissions is kept roughly $f + w$.

The purpose of the $FindNeighbour$ function is to find an alternate neighbouring time slot for frames originally scheduled in relatively 'crowded' time slots by algorithm NAIVE. This is achieved by letting a frame 'stray' backward or forward from its scheduled time slot.

At this point, we need to distinguish between advancing a frame in time and delaying it. Advancing a frame schedules it before it is due, wasting some bandwidth. On the other hand, delaying a frame increases initial delay for clients that expect it. Bandwidth wastage due to advancing a frame decreases as the frame number increases, since the fractional effect of this advance diminishes with increasing inter-frame gap. On the other hand, average initial delay due to delaying any frame increases since more clients wait for later frames. Therefore we use two parameters δ_b and δ_f so that the limits for shifting frame f from time t are $t - \delta_b(w + f)$ and $t + \delta_f \times w$. Reasonable values are $\delta_b \approx 0.05$ and $\delta_a \approx 0.1$.

Given these limits, there are multiple ways to implement the $FindNeighbour$ function. Some examples are:

L-R-SCAN: Starting from t, scan first left from t to $t - \delta_b(f + w)$ and then right from t to $t + \delta_f \times w$, looking for time slots with available bandwidth.

R-L-SCAN: Similar to L-R-SCAN, but start off going right.

SPIRAL: Do a distorted Spiral, alternatively going left and right, so that $t - \delta_b(w + f$ is evaluated just before $t + \delta_f \times w$.

```
for (f ← 1; f ≤ n; f ← f+1) {
   EstimatedBandwidth ← EstimatedBandwidth + 1/(f+w);
   for (t ← f+w; t ≤ t_max; t ← t+f+w) {
      if (ActualBandwidth[t] ≤ EstimatedBandwidth) {
         transmit(f, t);
         ActualBandwidth[t] ← ActualBandwidth[t] + 1;
      } else {
         t' ← FindNeighbour(f, t);
         transmit(f, t');
         ActualBandwidth[t'] ← ActualBandwidth[t'] + 1;
      }
   }
}
```

Figure 3: Algorithm BASIC

Further, it could find the instant with minimum bandwidth (Best fit) or stop at the first instant which can accommodate frame f (First Fit). [2] Finally, as a fallback, if all instants in the interval $(t - \delta_b(w + f), t + \delta_f \times w)$ exceed their allotted bandwidth, f is scheduled in the instant within this range with minimum bandwidth. However, our simulations suggest that for sane values of δ_b and δ_f, this seldom happens.

As shown in Figure 4(a), these strategies could be pictured as different paths from (t, t) to $(t - \delta_b(w + f), t + \delta_f \times w)$. SPIRAL could be thought of as the straight line path between the two points mapped by Bresenham's line algorithm. Extensive simulation over a wide range of parameters seems to indicate SPIRAL is a particularly robust way to implement $FindNeighbour$. Figure 5(a) shows bandwidth distribution for different $FindNeighbour$ functions. Figure 5(b) shows typical distribution of bandwidths for a 30 frames-per-second 2-hour movie for various values of initial delay w, with SPIRAL for FindNeighbour function.

As can be seen, the algorithm BASIC has one minor problem: When scheduling, we deal with frames as indivisible units. Therefore, if the theoretical server bandwidth is (say) 4.5,[3] the actual bandwidth varies equally between 4 and 5, resulting in peak bandwidths that overshoot the average by about 10%. However, this is easily remedied: If multiple movies are broadcast simultaneously (a likely scenario), they can be co-scheduled by modifying Algorithm BASIC to be aware of the actual and predicted global bandwidth when making scheduling decisions. Figure 6 shows that for as few as 8 streams, the peak server bandwidth usage matches optimal usage to within 2 percent.

Figure 7 compares bandwidth-delay performance of Fuzzycast with existing protocols and with the theoretical best performance.

Based on these results, we argue that Fuzzycast is a feasible broadcast scheme that offers optimal performance for on-demand streaming.

[2] For Best fit, the scan method does not matter.

[3] expressed in frames transmitted per instant

2.2 Layered Multicast

One problem with the basic scheme outlined in Section 2.1 is that every client continues to receive redundant transmissions, wasting client and network bandwidth. While this is unavoidable in a pure broadcast-based system[4], it is wasteful in a multicast situation where there is network support for subscribing to and unsubscribing from a multicast session. It is therefore desirable that each client explicitly deregister its interest in unwanted frame with the multicast infrastructure. The ideal solution of transmitting each frame in its own multicast group creates unacceptably high network overhead in the form of subscription/unsubscription messages and state information in the network routers.

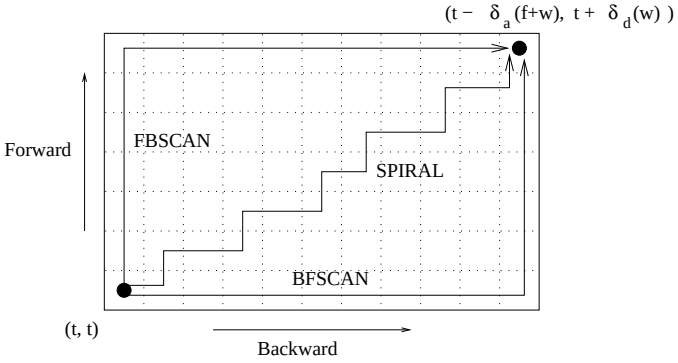
The solution is to use *Layered Multicast* (LM) [JMV96]. Most commonly, LM is used to deliver multiple versions the same transmission, differing only in quality, to heterogeneous receivers in a resource-efficient manner. Here, it is used to resource-efficiently deliver different time slots: Each client subscribes to all the groups on startup, shedding layers one by one as it is no longer interested in receiving data it already has received or played out.

An interesting problem in this connection is that of optimally splitting n frames into α groups. Making a group bigger increases cost in two ways:

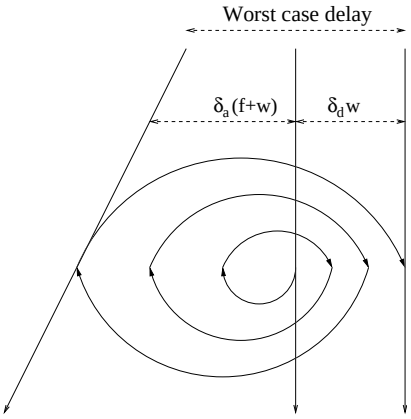
1. At any given time, more clients are subscribed to it. Even for broadcast mechanisms, this consumes more network resources, albeit sub-linearly.

2. Those clients are forced to listen to redundant transmissions longer.

[4] e.g., a satellite based distribution network

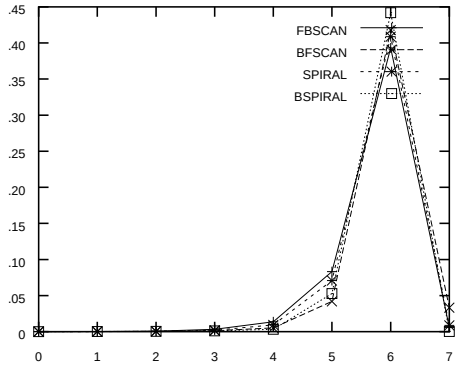


(a) Search Strategies

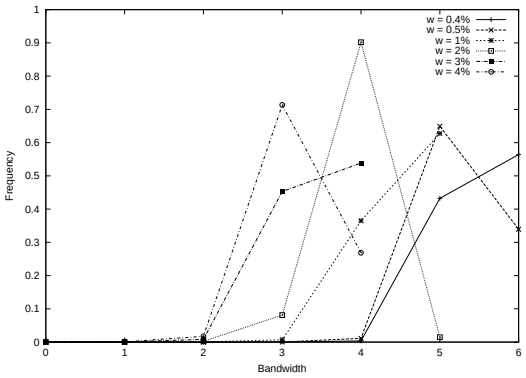


(b) Spiral FindNeighbour function

Figure 4: Optimal FindNeighbour Function



(a) Bandwidth distribution for different functions



(b) Bandwidth distribution for various w

Figure 5: Bandwidth Distribution

On the other hand, since the number of groups, α, is finite, dropping too early and too often would mean running out of groups sooner. As we shall see, there is an optimal solution.

2.2.1 Optimal allocation of Multicast Layers

Our first goal is to find the splitting that minimize the number of frames that each client receives. Consider putting group boundaries such that group k will last x_k instants and will contain frequencies $\frac{1}{x_{k-1}+1} \cdots \frac{1}{x_k}$ ($x_0 = w, x_\alpha = n + w$). The total number of frames that a client receives on an average is then given by:

$$\begin{aligned} F \quad &= x_1\left(\tfrac{1}{x_0+1} + \tfrac{1}{x_0+2} + \cdots + \tfrac{1}{x_1}\right) \\ &+ x_2\left(\tfrac{1}{x_1+1} + \tfrac{1}{x_1+2} + \cdots + \tfrac{1}{x_2}\right) + \cdots \\ &+ x_\alpha\left(\tfrac{1}{x_{\alpha-1}+1} + \tfrac{1}{x_{\alpha-1}+2} + \cdots + \tfrac{1}{x_\alpha}\right) \end{aligned} \quad (1)$$

In other words,

$$F \approx \sum_{k=1}^{\alpha} x_k \log \frac{x_k}{x_{k-1}} \quad (2)$$

Differentiating partially w.r.t. x_k and equating to 0 for minimum F, we get

$$x_{k+1} = x_k\left(1 + \log \frac{x_k}{x_{k-1}}\right) \quad (3)$$

Descending recursively, the first boundary x_1 is determined by,

$$x_\alpha = x_1\,(1+\log \frac{x_1}{x_0})\,(1+\log(1+\log \frac{x_1}{x_0}))\,\cdots(\alpha \text{ terms}) \quad (4)$$

where $x_\alpha = n + w$ and $x_0 = w$. This equation can be numerically solved (we use Newton-Raphson iteration) to get the

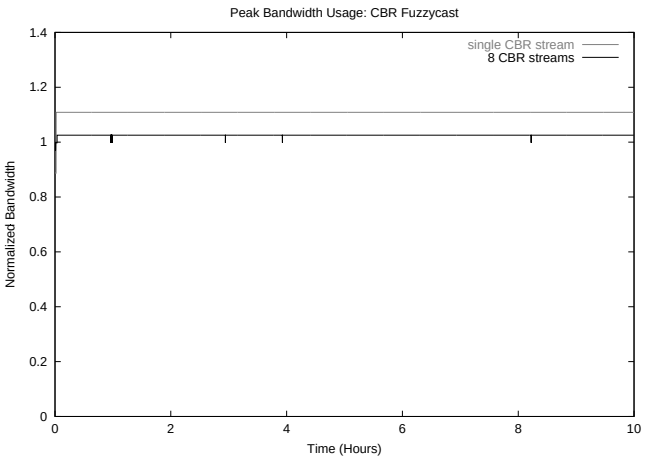


Figure 6: Bandwidth usage with Co-scheduling

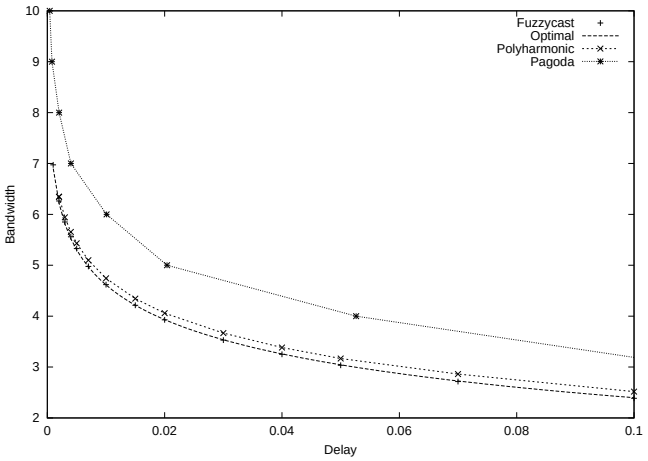


Figure 7: Performance comparison of Fuzzycast

value of x_1. Once x_1 is found, higher boundaries can be calculated using (3). This set of boundaries is the one that minimizes the number of frames that each client receives. For example for a two-hour, 30 frames-per-second movie, $\alpha = 3$ and $w = 30sec$, the optimal group boundaries are at 11.4 minutes and 49.1 minutes, leading to an average client bandwidth usage of 54 fps as opposed to 165 fps without layering. Figure 8 shows the reduction in bandwidth for various α. In the ideal case, when each frame is transmitted in its own group, average client bandwidth is exactly 1, so it is apparent that using even a small number of groups results in significantly fewer frames received by each client.

2.2.2 Minimizing Network cost

A connected problem is that of splitting n frames into α groups such that the overall network bandwidth is minimized. That is, we would like to minimize the total number of frames sent out

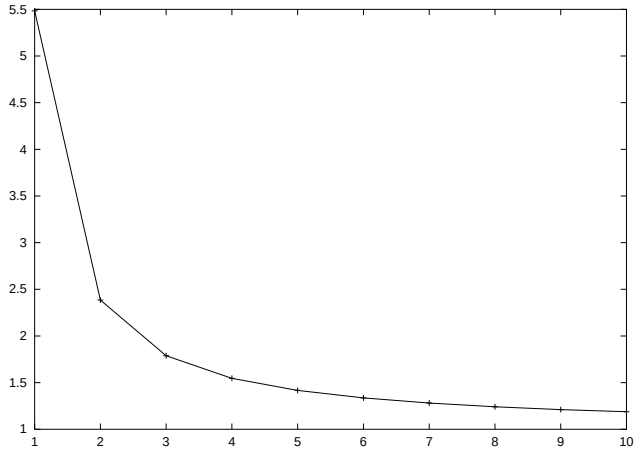


Figure 8: Average Client Bandwidth Vs. α

in the network at any time. If the number of links in a delivery tree of m clients is $L(m)$ and the average client arrival rate is λ then the number of clients subscribed to group k at any given time is $\approx \lambda x_k$. Therefore, the total network bandwidth is:

$$N \approx \sum_{k=1}^{\alpha} L(\lambda x_k) \log \frac{x_k}{x_{k-1}} \tag{5}$$

Previous work [CS98, PTS99] suggests that for Internet multicast, $L(m)$ is fairly accurately approximated by a power law of the form, $L(m) \approx \hat{u} m^{\rho}$ where $\rho \approx 0.8$ and $\hat{u}$ is the average unicast path length. This represents its network bandwidth advantage over multiple unicast, which has $L(m) = \hat{u}m$. The function to be minimized, then, is:

$$N \approx K \times \sum_{k=1}^{\alpha} x_k^{\rho} \log \frac{x_k}{x_{k-1}} \tag{6}$$

where $K = \hat{u}\lambda^{\rho}$ is a constant for a given transmission.[5] Proceeding as in Section 2.2.1, the analogy of (3) is:

$$x_{k+1} = x_k (1 + \rho \log \frac{x_k}{x_{k-1}})^{\frac{1}{\rho}} \tag{7}$$

For the same parameters as in Section 2.2.1, optimizing network bandwidth results in group boundaries at 9.24 minutes and 44 minutes for a network bandwidth $\approx 37\%$ of that without layering. Figure 9 shows the network bandwidth for various α, normalized to the bandwidth without layering.

2.3 An Adaptive Hybrid Scheme

Periodic Broadcasting is attractive for serving popular media because server bandwidth is independent of client arrival rates.

[5] For now, we assume constant λ. Section 2.3 describes a scheme with variable λ.

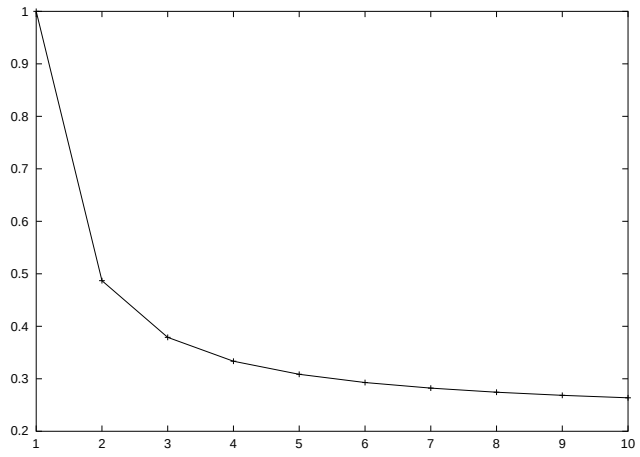


Figure 9: Network Cost Vs. α

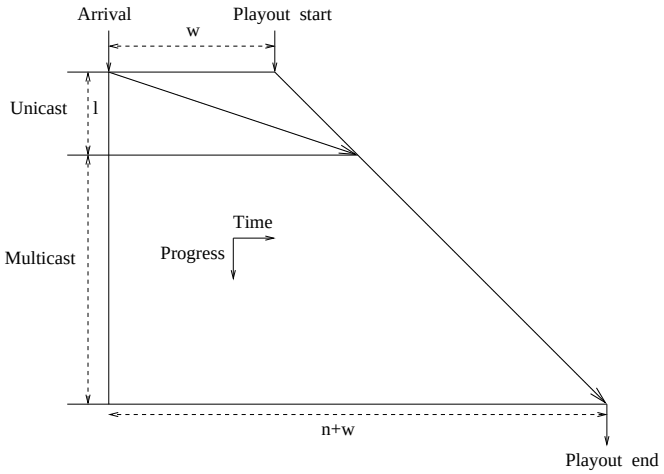


Figure 10: Hybrid Scheme

However, in reality, media cannot be classified as merely popular or unpopular: popularity occurs in a continuous spectrum. In broadcast schemes where different frames have different impact on the bandwidth, algorithms that classify media as either popular or unpopular end up making suboptimal use of bandwidth. Furthermore, user preferences change over time so that the set of "hot" videos is in a state of constant change. So, an adaptive, hybrid mechanism has the best chance of scaling well in an on-demand streaming system with widely varying client demands: adaptive, to handle changes in popularities gracefully; hybrid, to treat movies with widely different popularities uniformly.

In our hybrid scheme, instead of broadcasting frames $1, \ldots, n$ with frequencies $\frac{1}{w+1}, \ldots, \frac{1}{w+n}$, we unicast the first ℓ (say) frames to each client. Each client starts off listening to both the unicast prefix and the broadcast, starting playout after an initial delay of w instants. The unicast feed of ℓ frames is stretched over time $w + \ell$, leading to a unicast bandwidth of $\frac{\ell}{w+\ell}$ per client. This is illustrated in Figure 10. Our goal is to use broadcast for only those frames in which it would result in bandwidth savings.

2.3.1 Optimizing for Server Bandwidth

First, consider the choice of ℓ that minimizes server bandwidth. If client arrival rate is λ, unicasting frame f would take up a constant server bandwidth λ frames per instant, whereas broadcast would consume bandwidth $\frac{1}{f+w}$. Therefore, broadcast is more effective for frames f for which $\lambda \geq \frac{1}{f+w}$ or all frames $f \geq \frac{1}{\lambda} - w$. Thus the optimal value of ℓ that minimizes server bandwidth is $\frac{1}{\lambda} - w$.

This is illustrated in Figure 11. Some notable points are:

1. A unicast prefix is required only when λ falls below $\frac{1}{w}$, i.e., client inter-arrival time exceeds initial delay.

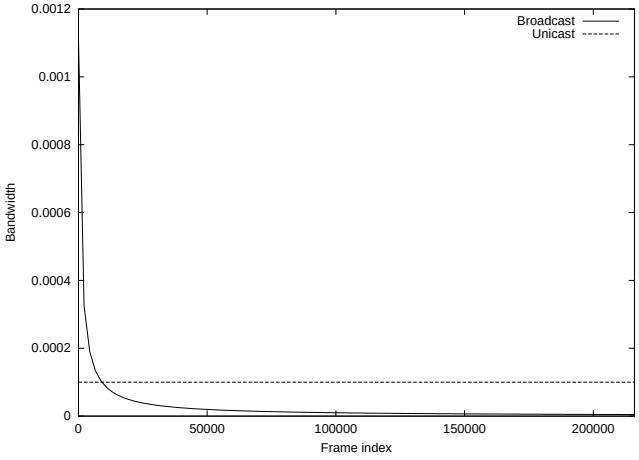


Figure 11: Bandwidth impact: Unicast Vs. Multicast ($w = 900$, $n = 216k$)

2. Decreasing λ increases ℓ, so that for $\lambda = \frac{1}{w}$, $\ell = 0$ (pure broadcast) and for $\lambda = \frac{1}{n+w}$, i.e., just one client for the entire movie, $\ell = n$ (pure unicast).

3. Each unicast lasts for $w + \ell$ instants and the number of concurrent unicasts at any time is roughly $\lambda(w+\ell)$. Thus, in the optimal case, at any given time, the expected number of clients receiving unicast is exactly 1.

Server Bandwidth usage S_{hybrid} for the hybrid scheme is:

$$\begin{aligned} S_{hybrid} &\approx \lambda\ell + \log \frac{w+n}{w+\ell} \\ &\approx 1 - \lambda w + \log \lambda + \log(w+n) \quad \text{(In the optimal case)} \\ &= 1 - \lambda w + \log(\lambda w) + S_{normal} \end{aligned} \tag{8}$$

where S_{normal} is the optimal bandwidth for the pure broadcast case.

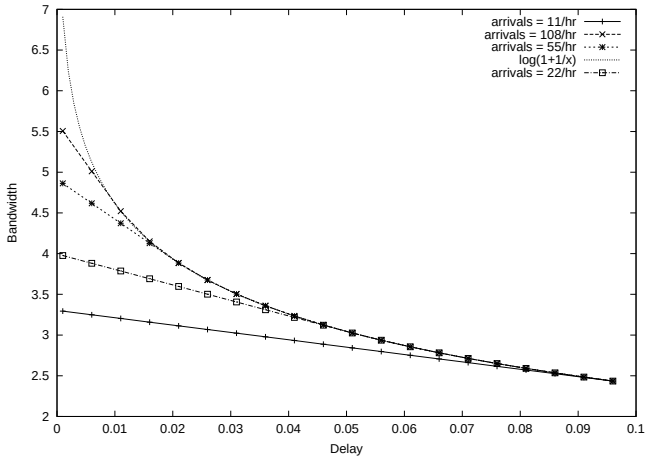


Figure 12: S_{normal} and S_{hybrid} for various client arrival rates λ plotted against initial waiting times w

The peak client bandwidth, C_{hybrid} is:

$$\begin{aligned} C_{hybrid} &= \frac{\ell}{w+\ell} + log\frac{w+n}{w+\ell} \\ &= 1 - \lambda w + \log(\lambda w) + C_{normal} \end{aligned} \tag{9}$$

Thus, in the hybrid scheme, client bandwidth is reduced by the same amount as the server bandwidth.

Figure 12 compares predicted S_{hybrid} and S_{normal} for different λ.

2.3.2 Network Bandwidth

When Layered Multicast is used for broadcast, adjacent frames belonging to different groups can have widely different impacts on the network bandwidth. In this case, the bandwidth impact in Figure 11 is no longer monotonically decreasing. Instead, we get a frame-impact graph as in Figure 13(b) Instead of a single unicast prefix as in Figure 10, we get multiple unicast streams, at most one for each layer, as in Figure 13(a).

2.4 Adaptive Scheme

The hybrid scheme in Section 2.3 works well for both popular and unpopular videos. But the optimal value of ℓ depends on λ. With variable client arrival rate, the system starts to drift away from optimality. The solution to this is to vary ℓ dynamically according to the client arrival rate at that time, i.e. increase it on decreasing λ and decreasing it with increasing λ. However, this introduces the complication of dealing with existing clients. Increasing ℓ cancels planned multicast schedules that these clients depend on. Decreasing ℓ is less harmful, since it merely schedules frames to be broadcast that existing clients do not really need by virtue of their longer-running unicast connection. Figures 14(a) 14(b) illustrate this. Assume clients arrive at times C_i arrives at time t_i, starts playout at time s_i and finishes at e_i, for $i = 1 \cdots 3$. Increasing the unicast prefix from ℓ_1 to ℓ_2 at time t_3 means that C_2 now has to follow the path $t_2 - A - F - e_2$ instead of $t_2 - A - E - e_2$. This means that the unicast bandwidth for all such clients should be increased to meet the broadcast at the new ℓ. However, this is not an option for clients like C_1, who are already done with their unicast part. This would mean that ℓ cannot be increased arbitrarily fast, but can only be increased without cancelling any frames that are scheduled within the playout of the latest client. This can be done aggresively, i.e. at evert instant t, if the latest client arrived k frames earlier, broadcast of frame $k - w$ can be canceled. Alternately, this can be done lazily on arrival of each client. In comparison, decreasing ℓ poses few problems. It schedules frame broadcasts that are already promised to clients by unicast, but since the server, with perfect knowledge of both the unicast and broadcast schedule, skips over unicasting frame f for client C if f is scheduled for broadcast before playout for client C. For example, the unicast path for client C_1, would follow a trajectory between $t_1 - A - B - C - e_1$ and $t_1 - A - C - e_1$, depending on how many of these frames it gets through broadcast.

2.5 Variable Bitrate

Till now, we have assumed our media is CBR. In practice, however, popular media like MPEG-2 or MPEG-4 are VBR, i.e., frame sizes are not constant. The basic scheme outlined above can be modified for VBR streaming by incorporating frame sizes into Algorithm BASIC. i.e for a n frame movie with frame sizes $f_1, f_2, \cdots, f_n$, the expected bandwidth for the first f frames is:

$$B_{VBR}(f) = \sum_{i=1}^{f} \frac{f_i}{w+i} \tag{10}$$

In conjunction with our global scheduling algorithm, this smooths bandwidth usage down to a great extent.

For example, Figure 15 shows the bandwidth usage [6] of 1-hour MPEG-4 movie streams, over a 10-hour period.

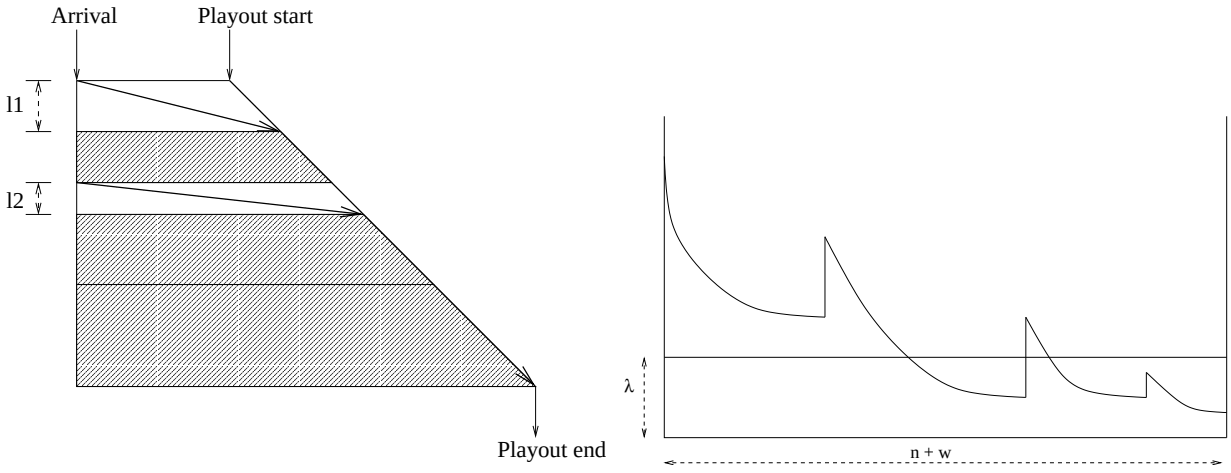
2.5.1 Fragmented Fuzzycast

As Figure 15 shows, Fuzzycast deals quite well with VBR sources as far as server bandwidth is concerned. However, dealing with VBR media is still undesirable because:

1. Client bandwidth does not benefit from the smoothing effect of multiple streams and suffers from significant bandwidth variability.

2. Having extremely variable frame sizes complicates buffer management, especially at the server side.

[6] Normalized to predicted bandwidth



(a) Multilayered Adaptive scheme

(b) Network Impact: Unicast Vs. Layered Multicast

Figure 13: Layered Multicast

	Network Support		Bandwidth	
Scheme	Feedback	Transmission	Server	Network and Client
Basic	no	broadcast	S_{normal}	S_{normal}
Layered Multicast	no	multicast	S_{normal}	$\approx \sqrt[\alpha]{S_{normal}}$
Hybrid	yes	broadcast	S_{hybrid}	S_{hybrid}
Layered Hybrid	yes	multicast	S_{hybrid}	$\approx \sqrt[\alpha]{S_{hybrid}}$

$S_{normal} = \log \frac{n+w}{w}$ $\quad$ $S_{hybrid} = S_{normal} - \lambda w + log(\lambda w) + 1$

Table 1: Scheme selection guide and simplified performance comparison

One way to circumvent the problem of VBR media is to 'pretend' that media is CBR i.e divide the content into constant size segments[7], cutting across frame boundaries. This might increase initial delays to an unacceptably high level, since bigger-than-average frames take multiple instants to arrive in their entirety. If frame f arrives completely by time t_f then initial delay is $\max\{t_f - f\} \geq w$, as shown in Figure 16.

An effective compromise is to use a smoothing mechanism like Piecewise Constant Rate Smoothing (PRCTT) [SZKT98, MR97], but this typically involves some increase in initial delay.

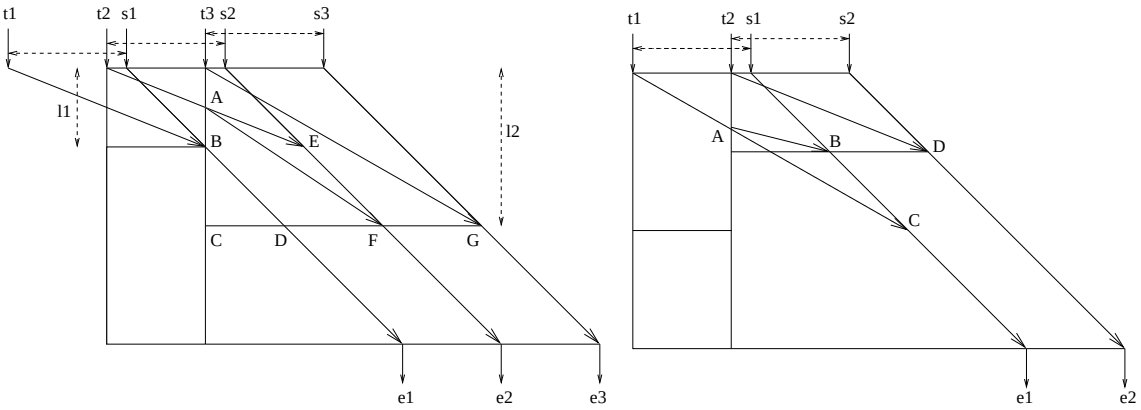
Fragmented Fuzzycast is a more effective solution: Consider a VBR encoded movie with a set of frames $F = \{f_1, f_2, \cdots, f_n\}$. Split it into a set of fixed sized blocks $B = \{b_1, b_2, \cdots, b_m\}$. Now, for each block b_i, there is a set $C(b_i) \subset F$ of frames which are either fully or partially contained in b_i. If the earliest frame in $C(b_i)$ is f_j, then transmit block b_i at frequency $\frac{1}{w+j}$.

[7] with size equal to average frame size

It is easy to prove that Fragmented Fuzzycast delivers all data on time. Block b_i is scheduled such that the earliest frame in it reaches on time. This implies later frames in the block also reach on time to every client, except possibly for the last one, which is truncated. However a frame truncated in b_i is the earliest frame in b_{i+1}, thus reaching on time. The last block has no truncated frames at the end, thus delivering all its frames on time.

As Figure 17 shows, the effect of fragmented fuzzycast is to replace a uniform playout line as in Figure 1 with a segmented one consisting of diagonals interspersed by vertical lines and horizontal lines. Vertical lines occur when a frame spans multiple blocks, horizontal lines when a block holds multiple adjacent frames.

Fragmented Fuzzycast is simple to implement, as Figure 18 shows: We just maintain pointers to the end of the current block in b_{block} and the current frame in b_{frame}, which grow at rates $blocksize$ and $size(j)$ respectively. Whenever the block pointer overtakes the frame pointer, the frame number is increased until this is reversed.



(a) Increasing ℓ (b) Decreasing ℓ

Figure 14: Adapting network bandwidth

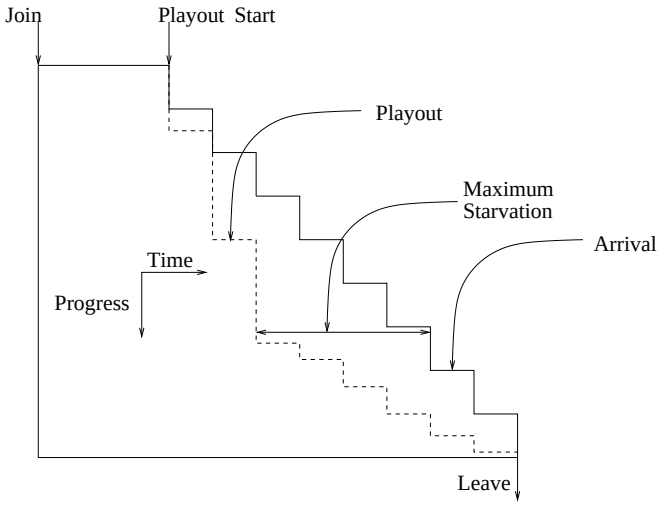


Figure 16: Starvation due to VBR media

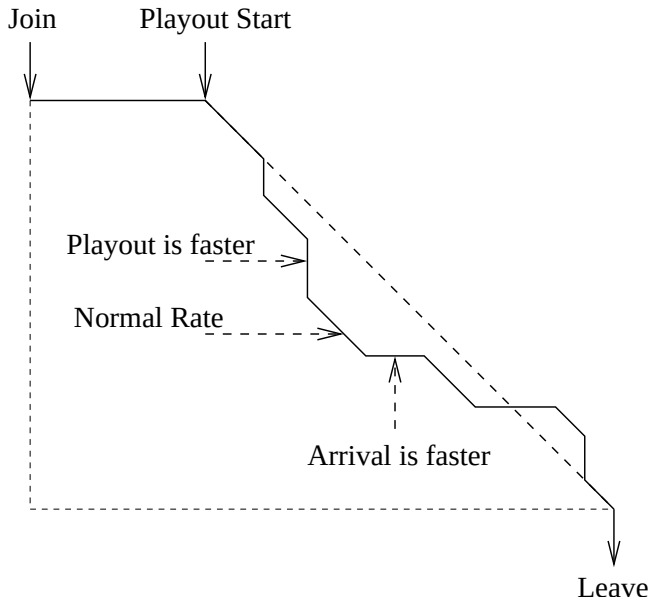


Figure 17: Fragmented Fuzzycast transmission

Figure 19 shows that Fragmented Fuzzycast is quite effective in smoothing rate variability of VBR traffic: The graph is virtually a duplicate of the CBR bandwidth usage graph Figure 6.

2.6 Implementation

Obviously, the more support from the network, the more efficiently network resources can be utilized. As summarized in Table 1, having a return channel from the receivers or a resource-efficient multicast capability will greatly help to reduce the bandwidth usage. The four simple and practical schemes that have been introduced in this paper, closely model the respective theoretical optima, for where they are known. We are confident that for the combinations of network support where bounds are not known yet, our schemes are close to optimal.

Our prototype implementation, comprising $\approx$ 500 lines of JAVA code[8], streams multiple movies over our campus network. Preliminary tests indicate that performance matches our predicted results.

Despite the sometimes rather complex formulae, the imple-

[8] for both server and client

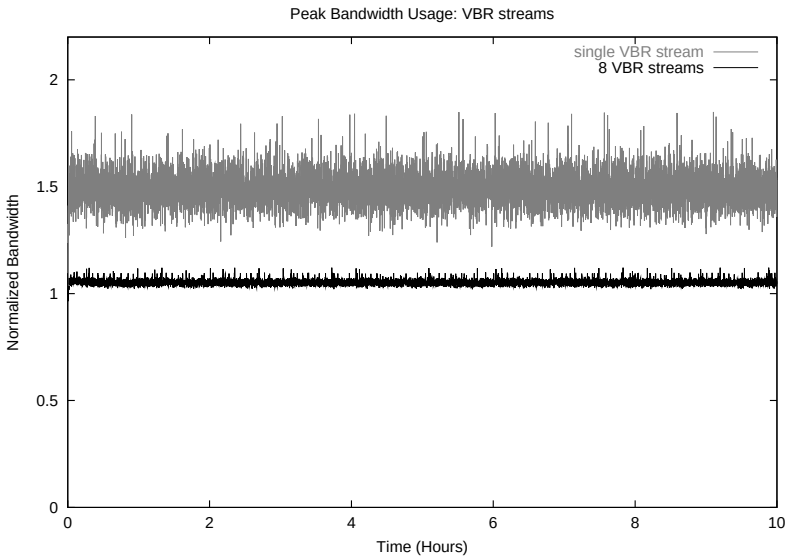


Figure 15: Bandwidth Usage for Fuzzycasting VBR streams

```
procedure Fragment
{
    b_block ← b_frame ← 0;
    foreach frame f_j
    {
        while (b_block ≥ b_frame and j < n )
        {
            b_frame += size(f_j);
            j++;
        }
        transmit next block at frequency 1/(w + j);
        b_block += blocksize;
    }
}
```

Figure 18: Fragmented transmission

mentation consists of simple components only: the BASIC and FINDNEIGHBOUR algorithms, a precomputed table of layer boundaries. Even for the adaptive schemes, keeping track of the join times of just the two most recent receivers is sufficient.

3 Client Disk Bandwidth Management

The scheme proposed in Section 2 involves segmentation and asynchronous broadcast of media data, which requires clients to efficiently buffer out-of-order segments and reorder them for serial playout. The conversion of out-of-order arrival to in-order playout suggests the use of external memory priority queues, but their content-agnostic nature prevents them from performing well under on-demand streaming loads. In the following we will propose and evaluate a series of simple heuristic schemes which achieve significant improvements in storage performance over existing schemes.

Buffer management could also be treated as a cache replacement problem, with the additional property that there are at most two accesses to each block, one random (during arrival) and the other serial (during playout). The optimal cache replacement strategy of writing out the frame with most distant playout to disk, while minimizing the number of disk accesses, ends up using one seek for every frame read or written, incurring a high overall disk I/O cost.

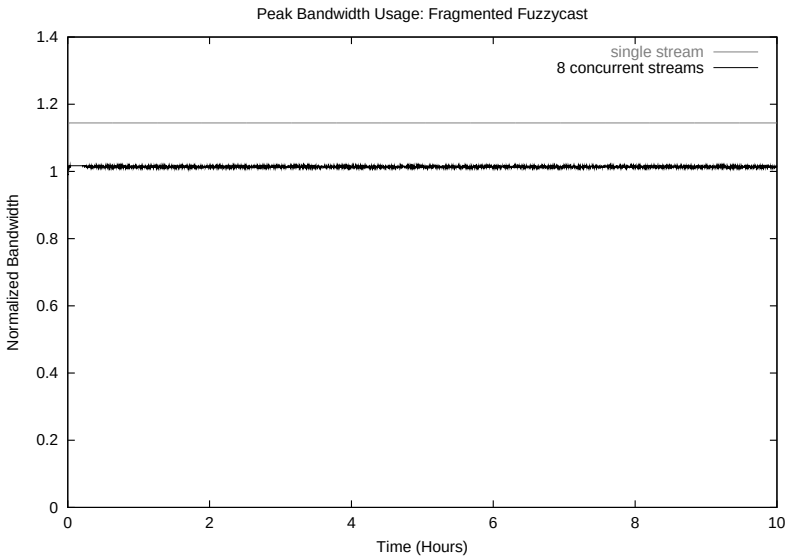


Figure 19: Bandwidth Usage for Fragmented Fuzzycast

3.1 Disk space

3.1.1 Expected disk space for Algorithm NAIVE

For a receiver that joins sometime in the middle, the frames arrive out of order and have to be cached on disk, to be played out in their time. The probability that frame f_i arrives in time j (relative to the join time) is:

$$p_i(t=j) = \begin{cases} \frac{1}{i} & 1 \le j \le i, \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

The probability that frame f_i is on disk during time j ($j < i$) is then:

$$p_i(t \le j) = \sum_{k=1}^{j} p_i(t=k) = \begin{cases} \frac{j}{i} & j < \text{i}, \\ 0 & \text{otherwise.} \end{cases} \tag{12}$$

So, the number of frames on disk at any given time j is simple:

$$\begin{aligned} d(j) = \sum_{i=1}^{n} p_i(t \le j) &= 0 + \sum_{i=j+1}^{n} p_i(t \le j) = \sum_{i=j+1}^{n} \frac{j}{i} \\ &= j \left\{ \sum_{i=1}^{n} \frac{1}{i} - \sum_{i=1}^{j} \frac{1}{i} \right\} \\ &\approx j \log \frac{n}{j} \qquad \text{(for large } j \text{ and } n\text{)} \end{aligned} \tag{13}$$

It can easily be seen that the peak disk space use is $d_{max} = \frac{n}{e}$, at time $\frac{n}{e}$ (e being the Euler's constant).

3.1.2 Disk space measurements for Algorithm BASIC

Figure 20 plots the disk space demand for several variants of algorithm BASIC. We find that the variation is not significant enough to choose one algorithm over another.

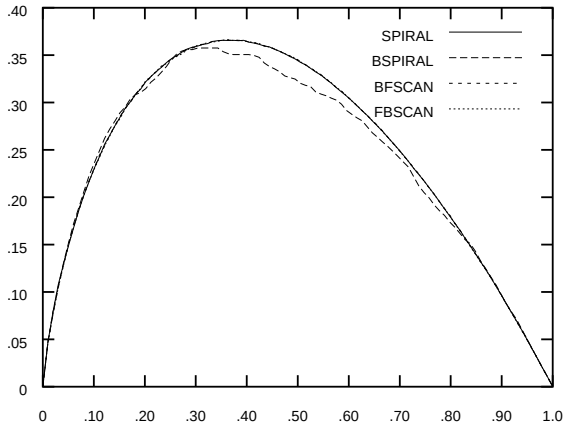


Figure 20: Disk space demand

3.1.3 Reducing disk space

Since video frames arrive largely out of order for a typical receiver, disk space requirements are large and peak at around 40% of the total length of the movie. This disk space requirement can be reduced at the cost of some bandwidth. Corresponding to a server using algorithm BASIC as the scheduler, the client grabs and holds on to the first instance of any frame

that comes along. If the client knows that another instance will come along before that frame is to be played out, it can afford to ignore the first. By transmitting later frames at a higher rate than absolutely necessary, we can trade a small increase in bandwidth for significant reduction in disk space demand. One way to do it is to ask the client to ignore frames $> v$ asking to be stored for more than λ instants. To make good on this promise, all frames $> v$ need to be transmitted every λ instants. A client, knowing v and λ, can safely ignore a frame f ($> v$) that it encounters at a time $< f - \lambda$. Arguing along the same lines as in Section 3.1.1, we get these disk space demand patterns:

$$d(j) = \begin{cases} j \log(\frac{v}{j}) & j < v - \lambda, \\ j \log(\frac{v}{j}) + \frac{(j-v+\lambda)^2}{2\lambda} & v - \lambda < j < v, \\ \frac{\lambda}{2} & v < j < n - \lambda, \\ \frac{\lambda}{2} - \frac{(j-n+\lambda)^2}{2\lambda} & n - \lambda < j < n. \end{cases} \quad (14)$$

The increased bandwidth is given by:

$$b = \log(v) + \frac{n - v}{\lambda} \quad (15)$$

Figure 21 gives the theoretical disk space demand over time, for 200k frames.

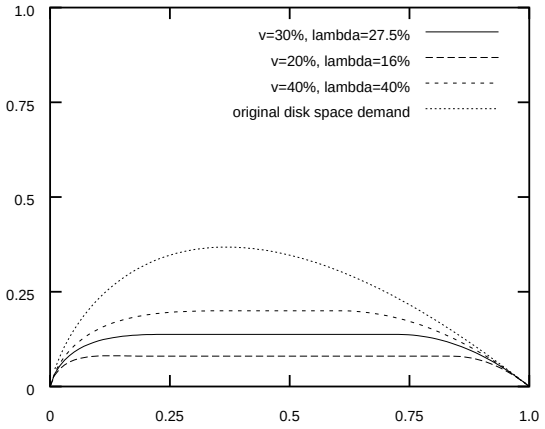


Figure 21: Reduced disk space demand

3.2 Efficient Buffer Management Schemes

From Section 3.1 we know that the peak buffer size is $\frac{n+w}{e}$ of the entire movie, at a fraction $\frac{n+w}{e}$ of the entire playout time into movie. For a two hour MPEG-2 movie with a transfer rate of 4Mb/s and a 30 second initial delay, this would translate to a peak buffer requirement of around 700 MB. Thus the typical client is forced to distribute its buffer between a small fast cache in primary memory and a large, slow hard disk that holds the bulk of the buffered frames. As frames arrive over the network, existing frames are displaced from the cache and written to disk till playout time. This naturally leads us to the question of doing this efficiently from a disk I/O performance perspective. Why is this important? Some motivations are:

- For set-top boxes, more efficient buffer management obviates need for higher performance hardware, leading to lower costs.
- Clients could be commodity PC's running multiple concurrent tasks: both memory and storage are shared resources that should be used optimally.
- Some proxies or transcoding devices located e.g. at cable head-ends receive, buffer and reorder frames (among other things) before streaming them serially to constrained end-systems like diskless set-top boxes. Better buffer management contributes to greater scalability.

3.2.1 Disk Metrics

We use a relatively simple metric for estimating disk performance: Consider a disk with these parameters:

RANDOM SEEK TIME, S_r: This is the average seek time for unrelated read/writes.

SEQUENTIAL SEEK TIME, S_s: This is the seek time for adjacent blocks of the same read/write.

BLOCK SIZE, B: Disk space is always allocated in blocks.

TRANSFER RATE, T: This is the rate at which data can be read/written.

If we assume that the disk allocator writes data in one write over multiple adjacent blocks, the time taken for a read/write of b bytes is given by:

$$t(b) \approx S_r + \frac{S_s b}{B} + \frac{b}{T}$$

Effectively, the time to transfer a block of n frames in one read/write between memory and disk is of the form:

$$t(n) \approx C_1 + C_2 \times n \quad (16)$$

where C_1 and C_2 are constant for a given disk. This model might seem simplistic in these times of intelligent caching disk controllers, but considering the massive amounts of data involved, our analysis shows that (16) provides a close approximation. For typical disk and transmission parameters, $C_1 \approx 10ms$ and $C_2 \approx 2ms$. We have arrived at these parameters by experimenting with simulations of the *Seagate Barracuda* disk, as obtained from DiskSim [Gan95]. Further, we are working on refining these measures, possibly through more complex disk models.

During the course of playout of the movie, a number of frames are written to disk and read back again. Our goal is to minimize total I/O time spent over these frames, as estimated by (16). In the following sections, we present our algorithms.

3.2.2 Most Distant Playout (MDP) Replacement

MDP is similar to the optimal cache replacement algorithm: the principle is to replace frames that would be required farthest in the future. With sequentially accessed media data, the highest numbered frame in cache is the ideal candidate for replacement.

In MDP, instead of replacing just the last frame in the cache, a number κ of frames in cache are written out to disk as a single chunk. Preemptively writing out a bunch of likely-to-be replaced frames amortizes seek time over κ frames, instead of using up a seek for each frame. When the earliest frame in the sequence is to be played out, the entire chunk is read back into cache again.

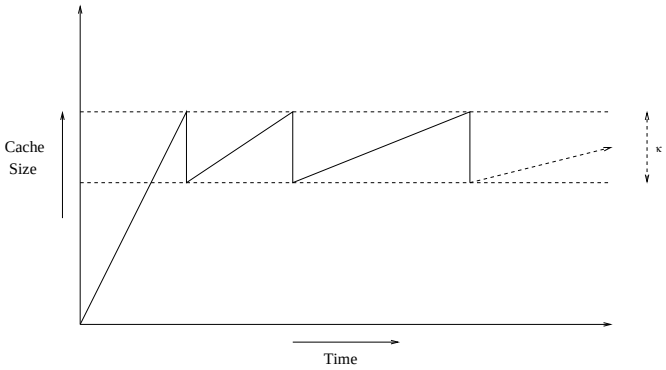


Figure 22: Buffer growth with batched writes (Schematic)

MDP is suboptimal in the number of total accesses to disk, some frames cycle more than once between memory and disk. But since access time for reasonable frame sizes is much lower then seek times, MDP, by effectively trading in more reads/writes for fewer seeks, is able to achieve a reduction in overall disk I/O time.

3.2.3 Most Compact Sequence (MCS) Replacement

Another strategy is to take advantage of linear access of frames and write out the κ long run of frames that has the most compact playout schedule of the frames currently in memory, in an attempt to reduce the rate of blocks cycling between disk and memory. For a run of κ frames with playout times $P_1, P_2, \cdots P_\kappa$, we define its *sparseness* (as opposed to compactness) by

$$S = \sum_{i=1}^{\kappa} (P_i - P_1)$$

We choose the rightmost such sequence with minimal S. In the best case, this is a stretch of κ continuous frames. Sparseness can be thought of as a cumulative measure of wasted buffer occupancy: the earliest frame in a sparse run wastefully drags

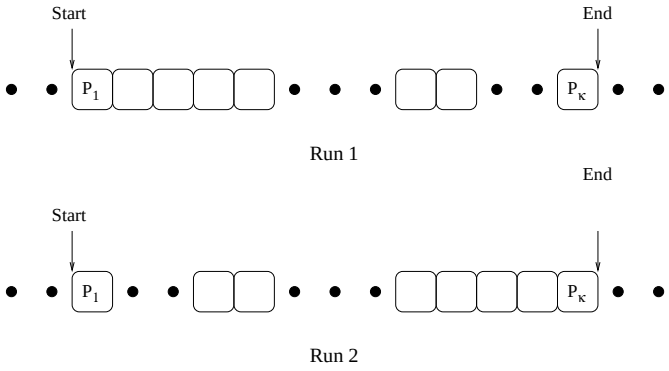


Figure 23: Two compact runs. Squares represent frames in memory, dots their absence.

along much later frames from disk to memory. Replacing compact runs lowers the risk of later frames in the run getting replaced again before playout.

An advantage of this definition is that it selects Run-1 over Run-2 in Figure 23, although both have the same 'density'. A run with more frames near the head is a better candidate for replacement because fewer frames (if at all) get cycled back to disk, while fetching other frames necessary for playout.

3.3 Optimal Block sizes

The rationale for writing out blocks of κ frames instead of single frames is to trade increases in cheaper disk transfers for fewer costly seeks. The optimal value of κ depends on the relative cost of seeks vs. read/writes as well as the cache size.

Increasing κ reduces seeks and buys lower disk I/O time upto a point. Thereafter, increased access times predominate and our technique is no longer optimal. Figures 3.3 and 3.3 plot κ as a fraction of C, the cache size against I/O time as a fraction of $n + w$, the playout time for MDP and Figure 3.3 for MCS. It can be seen that $\kappa \approx 2 \ldots 3\%$ of cache size gives best performance for both MDP and MCS. All I/O times are given relative to the actual playout time. We find that a seek would take roughly the same time as transfering 5 frames. As all of the blocks written to disk are of the same size, disk allocation management becomes trivial. We have also experimented with a variable κ, but we feel that the performance gains accrued might not be worth the increased complexity in disk management.

3.4 Performance Comparison

Figure 26 summarizes the performance of MDP and MCS replacement schemes in comparison with the radix heap based algorithm and the optimal cache replacement algorithm. In our implementation of radix heap algorithm presented in [BK98] , we have improved its efficiency somewhat by modifying it

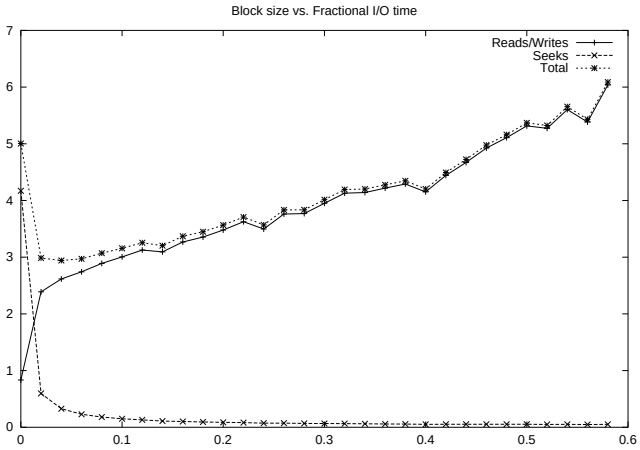


Figure 24: MDP: $\frac{\kappa}{C}$ vs. I/O time

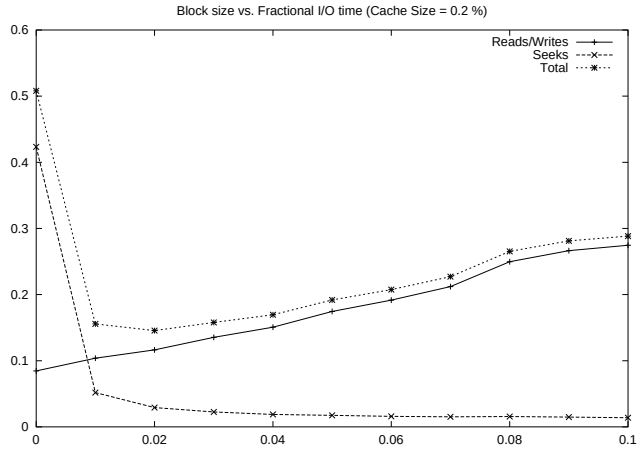


Figure 25: MCS: $\frac{\kappa}{C}$ vs. I/O time

to take advantage of all the memory available and write out buckets to disk only when this memory is used up.

It can be seen that in the feasible operating regions of Cache size = 0.2 % to 2 % ($\approx$ 6 to 60 MB for typical MPEG-2 movies), our schemes outperform existing schemes by more than a factor of two in disk I/O. Radix tries as described by the authors [BCFM99] would have performed an additional three times worse. Experimental results using array heaps [BCFM99], another promising data structure, performed no better.

3.4.1 Computational Complexity

We note that both MDP and MCS are quite easy to implement compared to heap based schemes, since they use relatively simple structures like lists or arrays. A simple array based implementation of MDP with a cache of size C frames takes $O(C)$ time to insert a frame and $O(1)$ time to delete κ frames from cache. Implementing the cache as a binary tree would

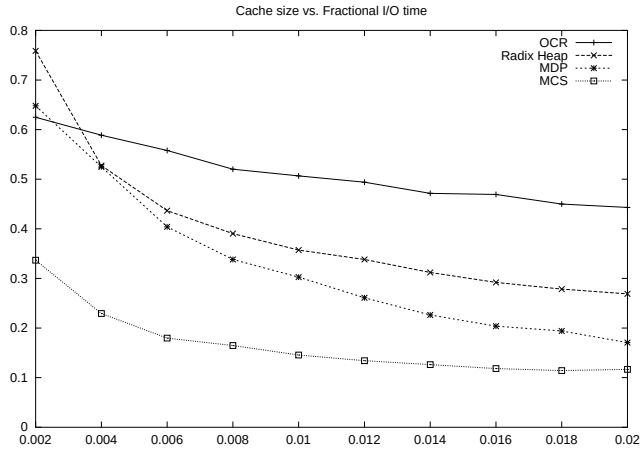


Figure 26: Cache size vs. I/O time

result in $O(log(C))$ time for inserting and deleting a frame. A simple array based implementation of MCS takes $O(C)$ time to insert a frame and $O(C)$ time to replace a run of κ frames. Since memory constraints will typically limit C to not more than a few hundred, this is not prohibitively expensive.

4 Conclusions

4.1 Summary

We have proposed a practical method to achieve near-optimal performance in on-demand media streaming. Our technique matches the performance of the best existing protocol while sharing none of its complexity and performs significantly better than the best existing scheme of comparable complexity.

Most current analysis has tended to focus on the server bandwidth. We have chosen a novel approach in using the Chuang-Sirbu law to minimize network cost. We are convinced that our technique will find broad application.

We also discuss an adaptive approach that uses an optimal mix of unicast and multicast, again using the multicast scaling law to find a solution that minimizes total network costs.

In our scheme, the client only needs about 37% ($\frac{1}{e}$) storage of the whole content, even in the worst cast. However, it is still a heavy load on the client in terms of buffer space. For designing cheaper end-systems or more scalable proxies, efficient buffer management assumes importance. We have proposed and evaluated two simple schemes drawing upon these principles:

BATCHED I/O: Cache replacement in blocks of multiple frames amortizes seek times over these frames. Empirically, we found replacing 2-3 % of the cache optimal.

USING ACCESS PATTERNS: Using knowledge of linear access in continuous media to replace compact sequences reduces cycling between memory and disk.

Together, these techniques achieve significant performance gains compared to algorithms described in literature.

4.2 Future Work

So far, we have addressed optimal server and network usages, and optimal average bandwidths at clients. Depending on the parameters chosen, we expect the ratio between average and peak bandwidth at the client to be in the range of $2 \ldots 5$. For clients with narrow network connections, this small factor might be enough to considerably reduce the maximum quality available to the viewer. We are thus looking at efficiently smoothing the peak bandwidth in the network, e.g., at DSL or cable head-ends. We are also working on making our scheme work with variable bit-rate media, employing some kind of smoothing mechanism like Piecewise constant rate smoothing. We believe this is a straightforward extension of our work.

Meanwhile, we are working on improving our heuristics and providing theoretical upper bounds for client disk bandwidth management. We are looking forward to finding out more about optimal cache replacement strategies when the server transmission schedule is known to the clients. Some of the algorithms might also be able to take more advantage of a variable κ.

References

[BCFM99] K. Brengel, A. Crauser, P. Ferragina, and U. Meyer. An experimental study of priority queues in external memory. In *Proceedings of the Workshop on Algorithm Engineering*, number 1668 in Lecture Notes in Computer Science, pages 345–358, 1999.

[BK98] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proceedings of the Scandinavian Workshop on Algorithms Theory*, number 1432 in Lecture Notes in Computer Science, pages 107–118, 1998.

[CS98] J. Chuang and M. Sirbu. Pricing multicast communications: A cost based approach. In *Proceedings INET '98*, 1998.

[DSS94] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Proceedings ACM Multimedia '94*, pages 391–398, October 1994.

[EVZ99] D. Eager, M. Vernon, and J. Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. In *Proceedings MIS '99*, October 1999.

[Gan95] Gregory Robert Ganger. *System-Oriented Evaluation of I/O Subsystem Performance*. PhD thesis, University of Michigan, Ann Arbor, 1995. Also available as technical report CSE-TR-243-95.

[HS97] K. Hua and S. Sheu. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. In *ACM SIGCOMM '97*, September 1997.

[JMV96] V. Jacobson, S. McCanne, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings ACM SIGCOMM '96*, pages 117–130, August 1996.

[JT97] L. Juhn and L. Tseng. Harmonic broadcasting for video-on-demand service. *IEEE Transactions on Broadcasting*, 43(3):268–271, September 1997.

[MR97] J. McManus and K. Ross. A dynamic programming methodology for managing prerecorded vbr sources in packet–switched networks. In *Proceedings SPIE, Performance and Control of Network Systems*, pages 140–154, November 1997.

[NBT98] Jörg Nonnenmacher, Ernst W. Biersack, and Don Towsley. Parity-based loss recovery for reliable multicast transmission. *IEEE/ACM Transactions on Networking*, 6(4):349–361, August 1998.

[PCL98a] J.-F. Pâris, S. W. Carter, and D. D. E. Long. Efficient broadcasting protocols for video on demand. In *Proceedings 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 127–132, July 1998.

[PCL98b] J.-F. Pâris, S. W. Carter, and D. D. E. Long. A low bandwidth broadcasting protocol for video on demand. In *Proceedings 7th International Conference on Computer Communications and Networks (IC3N'98)*, pages 690–697, October 1998.

[PCL99] J.-F. Pâris, S. W. Carter, and D. D. E Long. A hybrid broadcasting protocol for video on demand. In *Proceedings IS&T/SPIE Conference on Multimedia Computing and Networking 1999 (MMCN'99)*, pages 317–326, 1999.

[PTS99] G. Phillips, H. Tangmunarunkit, and S. Shenker. Scaling of multicast trees: Comments on the Chuang-Sirbu scaling law. In *ACM SIGCOMM '99*, 1999.

[SGT99] S. Sen, L. Gao, and D. Towsley. Frame-based periodic broadcast and fundamental resource tradeoffs. Technical Report 99-78, University of Massachusetts, Amherst, 1999.

[SZKT98] J. Salehi, Z. Zhang, J. Kurose, and D. Towsley. Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing. *IEEE/ACM Transactions on Networking*, 6:397–410, August 1998.

[VI96] S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. In *Multimedia Systems, Vol. 4*, pages 197–208, 1996.