# Peer-to-peer Streaming of Stored Media: The Indirect Approach

## ABSTRACT

We present the design and implementation of a stored media delivery architecture called Π-Stream. It allows clients to stream media directly from multiple servers and collaboratively from multiple other clients to support a rich media streaming service with high playback rate over the Internet. Π-Stream is a receiver-driven indirect streaming architecture which employs a software component called a local proxy stream server (LPSS) at the client. Such an architecture guarantees that the system works seamlessly with existing commercially available media streaming clients, requiring no change. Our prototype LPSS is fully implemented and tested on the wide-area Internet and demonstrated sustained playback rates of more than 2.4 Mbps. The software is currently available for public use from our website.

## 1. INTRODUCTION

The available network access capacity has increased significantly in recent years with more than 50% homes in US already having broadband connections [1]. This increased bandwidth has led to the proliferation of rich multimedia content that are accessed either from Content Distribution Networks (CDN) or through the file-sharing peer-to-peer networks. In fact, a recent study [2] shows that with larger amount of data being transferred, Intra-AS and peer links are becoming bottlenecks instead of the access links.

In order to circumvent the bandwidth bottleneck problem, the use of parallel downloads from multiple peers has gained popularity [3, 4] and is visible in peer-to-peer systems such as BitTorrent[5] and eMule [6]. In such a peer-to-peer (P2P) scenario, the content is usually replicated at multiple peer nodes (end hosts) and user downloads different blocks of the same content from different peers resulting in a faster download of the *entire* content. While such approaches are ideal for large file downloads, they are inadequate when applied to a *real-time* media streaming application (as demonstrated through our Internet experiments). This is because the goal of real-time media streaming is to enable fast commencement of playback without having to wait for the entire media to be downloaded first. This is very much unlike the structure of popular file sharing applications where each peer is attempting to download arbitrary bytes of the file from any other peer. Thus, while parallel file transfer techniques, e.g., eMule and BitTorrent, address the question of *what* to download from *which* server, a parallel media delivery architecture has to also decide *when* to download a block.

Today, all commercially available and publicly used media-streaming software, e.g., Windows media player, Real player, and QuickTime client, only work with a single streaming server. In particular, any such client will use the Real-time Transport Protocol (RTP) [7] / Real-Time Streaming Protocol (RTSP) [8] or some proprietary equivalent, and continuously receive the media stream from this specific server. While a protocol such as RTP has mechanisms to synchronize data from multiple sources (or servers), e.g., using the SSRC and TimeStamp fields in RTP packets, there exists no mechanism that provides a client the ability to partition the download across multiple such servers in a *fine-grained* manner governed by network dynamics. (Note that coarse-grained partitioning is possible by using a layered encoding scheme.) Consequently all current commercially available media-streaming clients restrict themselves to use only a single server. Hence they can neither take advantage of multiple available servers, nor can they leverage a peer-to-peer style design.

In this paper, we present Π-Stream, a system which enables fast real-time media-streaming that allows clients to take advantage of multiple servers, when available, as well as allows peer-to-peer style collaborations between such clients for improved performance. A key design objective in Π-Stream has been the following requirement: a user of Π-Stream should be able to use any existing (commercially deployed and widely available) media-streaming client software and get the above benefits. In doing so, the user should not need to make any changes to client software itself.

Π-Stream achieves this objective by breaking down the media delivery process into two independent stages by incorporating a *Local Proxy Stream Server* or LPSS between the set of server peers and the client software. The two independent stages are (i) an LPSS downloader that utilizes a simple protocol, such as HTTP, FTP, etc., to schedule downloads of different parts of the media content from the multiple servers and other clients as necessary; and (ii) an LPSS streamer that behaves like a streaming server and continually streams downloaded content locally in the client's media player in the desired playback format. The LPSS and the media player are both located in the client machine. We illustrate this in Figure 1. The servers in the figure are regular web servers that have the entire movie content. Each client in the figure is performing its own download and playback through its local LPSS. In this process each such LPSS is also opportunistically downloading some of the content from other LPSS nodes that it is aware of. Similar to BitTorrent, a LPSS finds other LPSS in the Internet (downloading the same media content) through a tracker service implemented by the content provider. The tracker maintains information about all LPSS nodes in the system. When an LPSS downloads the media from servers alone, we refer to it as an LPSS-standalone, and when it additionally downloads the parts of the media from other LPSS peers, we refer to it as LPSS-Torrent.

Using this design, Π-Stream not only provides a high degree of resilience to transient rate fluctuations and an ability to gracefully handle server failures, but also relieves the media clients from the burden of dealing with the vagaries of the network.

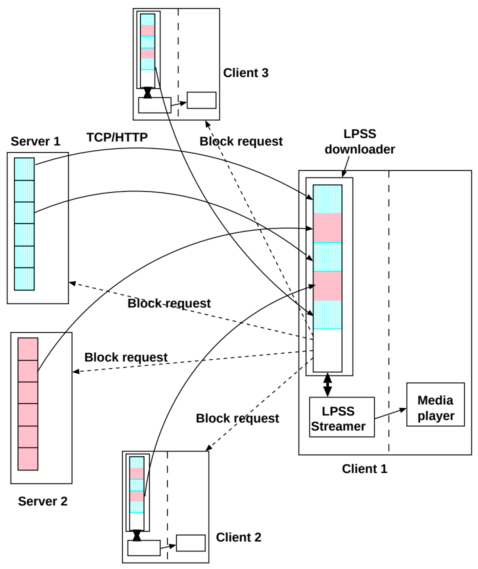

**Figure 1: The Π-Stream architecture.**

We refer to this approach of two stage media playback as *indirect streaming.*

## 1.1 Indirect Streaming

The fundamental advantage of indirection is that it decouples the media playback from media download. Thus, the media player need not be aware of *how, when or from where* the media arrived in its playback buffer. Indirection enables *protocol-independence* at both source and receiver requiring no modification to servers, video client application, or video encoding. For example, the LPSS can communicate with different servers using different protocols (HTTP, FTP) without the media client being aware of them with the LPSS acting as the communicating and translating media-hub. There is a significant advantage of such an indirect streaming design. In such a design, providers of streaming content can store media content in public web servers deployed in different locations and the LPSS will use the popular HTTP protocol to carefully schedule transfer of blocks in streaming order from these servers on-demand to play them back to the client. While a similar design is feasible with distributed media-streaming servers, it is easier and usually cheaper to deploy web servers. Also media streaming from distributed streaming servers would require significant coordination and synchronization between the real-time media streams. While some recent work, e.g., PALS [9] and PROMISE [10], have proposed such real-time streaming using distributed media streaming peers as sources, to the best of our knowledge, no such system is currently *widely* available (unlike web servers) that has been implemented and demonstrated in a wide-area setting. In contrast, our proposed media streaming approach can use any simple byte-range style block transfer mechanism implemented in HTTP or FTP and requires minimal coordination between the web servers. In fact, such coordination is implemented solely at the LPSS and requires no change to web servers.

Additionally, unlike direct streaming, where the rate is determined by playback rate and playback buffer, indirect streaming can use the disk storage and can download at a speed limited only by the access bandwidth. Since wasted bandwidth can never be reclaimed, Π-Stream uses extra storage to limit the bandwidth loss. Under most cases, users are interested in the entire media file for later use, therefore storage is hardly an issue. Finally, by not hosting the media server at peer nodes, they are relieved from the load of streaming and request handling load from control packets.

## 1.2 Distinction of Π-Stream from Prior Approaches

At a conceptual level, the P2P-based media streaming approach in Π-Stream is similar to that of PALS [9]. We consider both these approaches to be *receiver-driven,* whereby the streaming process is coordinated through decisions made at the client and not the server(s). However, the realization of Π-Stream differs from that of PALS in two significant ways. (i) Π-Stream, as described, employs indirect approach to media streaming. Instead of the client software directly fetching data from multiple sources, we design an independent software component, the LPSS, which performs this task. The LPSS interoperates with any existing streaming client that is publicly or commercially available and does not require any change to such software. In contrast, PALS can be viewed as an instantiation of the direct streaming approach and does not have the corresponding benefit. (ii) Π-Stream requires no special deployment of media-streaming servers in the Internet. Instead, it is able to seamlessly function using any data delivery protocol from the servers to the LPSS including any real-time media streaming standards like RTP and byte-stream approaches such as HTTP. In our implementation, we use Apache web servers (again without any modification) as sources and use HTTP byte-range request based media-streaming that operates over TCP. Clearly such an approach requires no additional congestion control mechanisms. This can be contrasted to the PALS approach which uses multi-source real-time streaming over UDP and hence requires the deployment of specialized media-streaming servers as well as that of relatively new congestion control mechanisms such as RAP [11] or TFRC [12].

The following are some of the key contributions of this work:

- Advocating the indirect approach to stored media-streaming for real-time playback in order to allow existing widely deployed client software to avail advantages of multiple servers and peer-to-peer style collaborations.

- Design and implementation of a functional prototype LPSS using this approach and demonstration of its use. In our experiments we used this prototype to enable media streaming to a widely available media-streaming client, QuickTime. No changes to the QuickTime client were necessary.

- LPSS, the proxy software in Π-Stream is made available in our website for public download experimentation, and use. (A current version is available from (link removed for double-blind review)and we expect to make it available through Sourceforge fairly soon.)

## 2. Π-Stream ARCHITECTURE

There are three different components in the Π-Stream architecture:

- Client: This includes the LPSS and the media player that are co-located in the same machine as independent processes.

- Server: Any byte-stream server, e.g., a web server or a FTP server, which stores the entire media content and serves these clients in small units of blocks on demand.

- Tracker: Analogous to the tracker used in BitTorrent, the tracker in Π-Stream provides information to each client about the location and the download status about various other clients in the system that are requesting playback for the same media content.

The media itself is split into blocks (segments) and each block can be downloaded independently. We represent the $i^{th}$ block as $B_i$ with its length denoted as $L_i$. The encoding is assumed to be constant bit rate (CBR) at a bit-rate of $r$, so that downloading the initial $x\%$ of a block $B_j$, corresponds to an expected playback duration of $x\%$ of $L_j/r$. The playback starting time of block $B_j$ is denoted by $s_j$, its finish time is $f_j$ and the two are related as $f_j = s_j + \frac{L_j}{r}$. Each block represents the *pre-specified* unit of transfer for the system.

A media player initiates the playback by sending an appropriate request to its local LPSS. The LPSS in turn contacts the appropriate tracker. The tracker responds with location information of a few (web) servers that store the relevant content file. Additionally the tracker also informs the requesting LPSS about the location of other peer LPSS nodes involved in playback for the same content.

In our implementation, we also assumed that each block is transferred from peer node using TCP. We define a download buffer (DB) possibly on disk that stores the downloaded blocks. The LPSS is responsible for all communication with the servers and other peer LPSS nodes and hides the network dynamics from the media player. It contains the block scheduler (described later), the rate measurement component, and has the exclusive write access to the DB. The LPSS hosts a stream server that continuously reads the DB, performs media format translation if needed and streams the media to the client media player.

Given a set of servers and other peer LPSS nodes, the job of the requesting LPSS is two-fold. First, it needs to appropriately identify a relevant subset of servers and other peer LPSS from which it should schedule the download. Unlike BitTorrent, the choice of other peer LPSS nodes depends not only on the data they possess, but also on how far ahead in the media playback these nodes are in the download process, their transfer rates to the requesting LPSS, and how long these LPSS nodes are expected to be around in the system. We call this the peer/server selection algorithm. Second, given a set of chosen servers and peer LPSS nodes, the requesting LPSS needs to plan a download schedule of different blocks and adapt this schedule with changes in network conditions. This is the block scheduling algorithm.

The peer/server selection algorithm is continuously run in a coarser time granularity to select the relevant set of servers and peers, while the block scheduling algorithm is run at a finer granularity to download blocks from the currently chosen set of servers and peers.

**Variables:**
$B_i$ - Set of blocks $i = 1, 2, \ldots, N$
$L_i$ - Length of block $B_i$
$Server_i$ - Connection assigned to block $B_i$
$\mathcal{R}(i)$ - Feasible connection set for $B_i$
$r$ - CBR playback rate
$s_i$ - The playback start time of block $B_i$
$f_i$ - The playback finish time of block $B_i$
$R_j^*$ - Sorted rates of Connections $j = 1, 2, \ldots, K$
$\gamma_j$ - Connection-id of $j^{th}$ fastest connection
$\beta_j$ - Busy time of $j^{th}$ fastest connection
$\mathcal{B}(i,j)$ - Busy time of $j^{th}$ fastest connection just before the instant when $B_i$ is assigned some connection

**Figure 2: The list of variables used in the algorithms.**

We note that the use of large blocks for download is a side-effect of our design tenet that existing infrastructure be used for incremental deployment. Our implementation uses web-infrastructure by identifying blocks using the byte-range request headers of the HTTP protocol. Hence, if the requests are issued at a fine granularity, the overhead of adding HTTP header to the reply for each packet is enormous. So it is a must to have large size blocks for efficient operation of the system (the choice of block size is discussed later). This mandates that we design an appropriate scheduling algorithm which determines which blocks to download from which servers.

In the next section we will first describe the download scheduling process (block scheduling algorithm) and subsequently in Section 4 explain how the choice of peer LPSS and servers are made by the requesting LPSS (peer/server selection algorithm).

## 3. BLOCK SCHEDULING ALGORITHM

We refer to a connection to server (or peer LPSS) $j$ as $C_j$ and the connection's throughput to the LPSS at time $t$ as $R_j(t)$. All $R_j(t)$s are assumed to remain quasi-static in the short duration (possibly on the order of a few RTTs) but vary over a long term. The $R_j(t)$s are computed using synchronous exponential averaging as $R_j(t) = \alpha * R_j(t-\delta) + (1-\alpha) * (Rate\ during\ last\ \delta)$ where $\alpha$ is the averaging parameter and $\delta$ is the duration of the update interval. At any time $t$, $R_1^*(t), R_2^*(t), \ldots, R_K^*(t)$ represents the sorted list (in decreasing order) of the TCP transfer rates $R_1(t), R_2(t), \ldots, R_K(t)$. At all times, $\gamma_j$ represents the index of the connection which has the $j^{th}$ fastest connection.

The LPSS contacts the servers and peers and requests one block from each of them. The initial assignment of blocks to servers or peers is sequentially done and continues in the playback buffering stage. At this stage, initial connection bandwidth measurement is done as well. Based on these measurements and playback deadlines, the LPSS constructs the download schedule for each block and assignment of these downloads to corresponding servers. This schedule and assignment gets re-adjusted with time based on bandwidth fluctuation.

We now describe our adaptive scheduling algorithm to assign blocks to servers. It works toward minimizing the probability of a block not being available at its playback

```
Scheduling Algorithm
1.  For j = 1 to K
2.      $\beta_j \leftarrow \frac{Remaining\ Block(\gamma_j)}{R^*_j}$
3.  End For
4.  For i = 1 to N
5.      $\mathcal{R}(i) \leftarrow Feasible\_Set(B_i)$
6.      If $\mathcal{R}(i) == \phi$
7.          $temp \leftarrow Swap(B_i)$
8.          If $(temp == 0)$
9.              $temp \leftarrow Split(B_i)$
10.             $N \leftarrow N + 1$
11.         End If
12.         $i \leftarrow temp$
13.         For t = 1 to K
14.             $\beta_t \leftarrow \mathcal{B}(i, t)$
15.         End For
16.         If (swap) goto 6 else goto 5
17.     Else
18.         For t = 1 to K
19.             $\mathcal{B}(i, t) \leftarrow \beta_t$
20.         End For
21.         $z \leftarrow j$ having $min(\beta_j + \frac{L_i}{R^*_j}) \mid \gamma_j \in \mathcal{R}_i$
22.         $Server_i \leftarrow \gamma_z$
23.         $\beta_z \leftarrow \beta_z + \frac{L_i}{R^*_z}$
24.     End If
25. End For
```

**Figure 3: Algorithm to assign connections to blocks if the connection transfer rates are static over the entire playback duration.**

```
Feasible_Set($B_i$)
1. $Set \leftarrow \phi$
2. For j = 1 to K
3.   If $((\beta_j \le s_i) \&\& (\beta_j + \frac{L_i}{R^*_j} \le f_i))$
4.     $Set \leftarrow Set \cup \gamma_j$
5.   End If
6. End For

Swap($B_i$)
1. For j = 1 to k
2.   $\beta_{req}(j) \leftarrow max(\beta_j - s_i, \beta_j + \frac{L_i}{R^*_j} - f_i)$
3. End For
4. For j = 1 to i − 1
5.   $R_{temp} \leftarrow Rate(Server_j)$
6.   $\beta_{temp} \leftarrow \beta(Server_j) - \frac{L_j}{R_{temp}}$
7.   If $(\beta_{temp} \ge \beta_{req}(Server_j))$
8.     If $(\mathcal{R}(j) - Server_j) \neq \phi$
9.       $\mathcal{R}(j) \leftarrow \mathcal{R}(j) - Server_j$
10.      return j
11.    End If
12.  End If
13. End For
14. return 0

Split($B_i$)
1. $T_{max} \leftarrow$ block with $max_{j=1}^{i-1} \frac{L_j}{Rate(Server_j)}$
2. $T_1 \leftarrow$ First half of $T_{max}$
3. $T_2 \leftarrow$ Second half of $T_{max}$
4. Recompute start and finish times of $T_1, T_2$
5. Renumber blocks starting from $T_1$
6. return index of $T_1$
```

**Figure 4: Algorithms to choose the feasible set of connections for any block, swapping the blocks and block splitting.**

time. The algorithm also tries to minimize the request load on servers. LPSS calls this algorithm in two situations: 1) When the rate monitor finds that a block being downloaded might miss its playback deadline, 2) when a server finishes its assigned block and LPSS has to assign a new block to that server.

The block scheduling algorithm takes the current estimated transfer rates $R_1(t), R_2(t), \ldots, R_K(t)$ and the estimated remaining blocks for each of the servers as an input. We omit the parameter (t) for clarity, since the rates do not change during the execution of this algorithm. The transfer rates are computed by the rate monitor and LPSS keeps track of the remaining data from each of the servers. Using this information, the block scheduler calculates the busy-time $\beta_j$, of the servers.

The block scheduling algorithm has to perform two important tasks: 1) it has to find a suitable block to assign to the free server and 2) it has to check whether the blocks within the look-ahead window (in the foreseeable future) have some feasible server assignment (after accounting for the times the servers would be busy downloading their currently assigned blocks.)

In solving the block transfer scheduling problem, we employ the following approach: 1) Get a given block at the earliest possible time subject to all previous blocks arriving at the earliest and its own playback deadline requirements being met, 2) Try to get any block in its entirety in a single request from one server, 3) Ask for sub-blocks only if the block's deadline is not likely to be met if it is downloaded as a whole. With this approach, the algorithm for assignment of blocks to servers is given below.

## 3.1 Earliest Finish Assignment

An obvious solution for block scheduling is the *Earliest Finish Assignment*. The key rule behind this algorithm is that we would like to get an earlier playback block at the earliest, subject to the condition that all the previous blocks arrive at their earliest. The above rule will lead to having the maximum cumulative amount of data at any given time or lexicographically (in block-ids) smallest finish time schedule. Note that this naive strategy is essentially the well-known earliest deadline first algorithm with the deadlines being determined by the block playback times. This basic algorithm is a subset of the one shown in Figure 3, specifically, it is the portion left after eliminating lines 6–20 (and also eliminating the EndIf statement in line 24). The algorithm is shown to assign the connections to *all* the blocks even though they might be downloaded from that server only at some distant future. This achieves the objective of testing feasibility for future blocks.

The earliest finish algorithm would be able to find a feasible schedule for all blocks only if each block has at least one server available in its feasible set (defined as the set of servers from which the block can be downloaded before its deadline given their download schedules of other blocks). However, this is not realistic if all the connection rates are not faster than the playback rate. Hence, we propose two additional strategies, swapping and splitting, to compensate for the limitations of the naive earliest finish assignment algorithm.

## 3.2 Connection Swapping

Since we aim at downloading a block as a whole from a server, the first option we look at is connection swapping. The idea behind connection swapping is to exploit any delay margin that the earliest finish assignment algorithm leaves. For example, consider a case where the earliest finish algorithm assigns a connection to a block which downloads 10 seconds before its playback start and 12 seconds before its finish. Thus, if we download the block after a little delay (say 5 seconds before start and 3 seconds before finish) by assigning it to a slower (and possibly busier) server, it would still suffice for the playback purposes. The advantage of this reassignment is that the original (faster) server would have a lower busy time and could help a later block by becoming the sole member of its feasible set. Thus, we could populate a block's empty feasible set by reassigning some previously assigned connections *while still being able to download the blocks at the specified granularity and meeting their deadlines.*

The algorithm for swapping is shown in figure 4. We use the original earliest finish assignment algorithm until the point that we find a block for which there is no feasible connection (after accounting for the existing assignments). Consider block $B_i$ to be the first block that has an empty feasible set $\mathcal{R}_i$. We try to find a suitable block starting from $B_1$ to $B_{i-1}$ with which if the server for $B_i$ were swapped, *all* blocks from $B_1$ to $B_i$ would have a feasible server.

Note that the swapping algorithm used here is a simple heuristic and does not cover all possible combinations (to avoid time complexity) of rate assignments to try and meet a block's deadline. One should note that the swapping algorithm works recursively toward finally meeting the deadlines. Lastly, it is possible that no swapping operations reach a feasible server assignment for every block. In such a case, we adopt the block splitting strategy described next.

## 3.3 Block Splitting

The insight behind block splitting (Figure 4) is that the granularity of busy times of a connection is at the level of a block. So if a large block is stuck with a slow connection, it would take a long time to download. We observe that if this block were divided into two smaller blocks, they could be downloaded in parallel using two separate connections. Effectively, block splitting allows us to increase the transfer rate assigned to the original block. However, as discussed earlier, small block sizes results in a large overhead due to which we use splitting as a last resort to handle adverse network conditions.

The splitting algorithm is shown in Figure 4. We choose the block with maximum download time as the block to be split. The reason being that breaking up this block could provide the maximum amount of time gained in terms of server busy time. We break this block into two halves and recompute the feasibility sets of the blocks to see if the new set of blocks has a feasible server assignment. The process is repeated till we get such a feasible assignment.

## 3.4 Other Issues

There are several important practical points are relevant to this algorithm that we considered in our implementation. Having too large a look-ahead could result in excessive block splitting particularly when the connection rates reduce drastically due to congestion. Our implementation provides the capability to re-merge the un-assigned sub-blocks or contiguous sub-blocks from the same server. The transfer rate of a pair of nodes can be measured using low overhead probes and measurements such as TFRC [12]. Alternatively it can be done actively, using short transfer bursts to check data rates. In our implementation we use a two-connection optimization (described in Section 5) for managing TCP connections. We use the 'passive' TCP connection to check data rates in short bursts.

## 4. PEER AND SERVER SELECTION ALGORITHM

We now proceed to describe the process of selecting a subset of servers and peer LPSS nodes from which the blocks are downloaded using the algorithm described in the previous section. With respect to a given LPSS, there are three key parameters that decide the choice of peers and servers. An obvious parameter is the sustained transfer rate between the peer (or server) and the requesting LPSS. However, as significant prior research has shown [13, 14, 15, 16], when conducting parallel downloads from multiple sites, shared congestion on the paths can play a significant role in choice of sources, rather than just the independent transfer rate.

Additionally there are two parameters when making such choices for peer LPSS nodes. This includes (i) the difference in playback time between the requesting LPSS and the peer LPSS — the further ahead the other peer LPSS is in the download process, the greater is the amount of additional data it can provide to the requesting LPSS; and (ii) the duration of time the other LPSS is expected to stay in the system — a peer that is expected to disappear from the system quickly is expected to be less useful to the requesting LPSS.

If we assume that there are no shared points of congestion on the Internet paths, then the problem of server (peer) selection can be performed using a simple, yet intuitive, heuristic as follows. (We consider the source to be another LPSS peer. The case when the source is another server follows analogously.) Let the requesting peer be labeled $P_1$ and a source peer be labeled $P_2$. Let $r_1$ denote the aggregate received rate of $P_1$ without having chosen $P_2$ as a source node. Let $r_{1,2}$ denote the possible rate achievable between $P_1$ and $P_2$. Let $r_2$ denote the aggregate received rate of $P_2$. (If $P_2$ is a server, then $r_2$ is 0.) Also let $\beta_1$ and $\beta_2$ indicate the (contiguous) bytes already downloaded by the two peers.

Now consider the case where $P_1$ chooses $P_2$ as a source of data. In general, if $r_1 + r_{1,2}$ is higher than $r_2$, then potentially $P_1$ will catch up with $P_2$ after a time $t*$, given by:

$$(r_1 + r_{1,2})t^* = r_2.t^* + (\beta_2 - \beta_1)$$

That is

$$t^* = \frac{\beta_2 - \beta_1}{r_1 + r_{1,2} - r_2}$$

In such a case, the total useful bytes downloaded by $P_1$ from $P_2$ is given by $r_{1,2}t^*$.

If, however, $P_2$ leaves the system at time, $t'$, prior to $P_1$ catching up with it e.g., if $r_1 + r_{1,2}$ is less than $r_2$, the total useful bytes downloaded by $P_1$ from $P_2$ is given by $r_{1,2}t'$.

Therefore, among multiple alternate choices in a set of self-congesting peers, we always choose a peer that can provide the highest amount of data to the requesting peer. This is given in either case by $r_{1,2}t$, where $t = t^*$ if $P_1$ catches up with $P_2$, and $t = t'$ otherwise. This simple heuristic, is iteratively evaluated in the long term to continuously update the selection of peers from which to download blocks from.

Note that the proposed heuristic is simple to implement, since the current aggregate download rate of each LPSS is reported and made available in the tracker. Based on this information and short bandwidth tests between a pair of candidate nodes the appropriate peer (server) selection choices can be made.

In presence of shared congestion on Internet paths, the above heuristic may not be directly applicable. In particular choice of one particular peer which shares a bottleneck link with another may preclude choosing the latter. There are numerous examples of shared congestion detection in prior literature. In our work we have experimented with one of them, by Rubenstein et.al. [13]. While the details of their proposed algorithm can be found in their paper, it is sufficient in this work to re-iterate the basic intuition of such shared congestion detection. To quote the authors " Informally, the point of congestion (POC) for two flows is the same when the same set of resources (e.g., routers) are dropping or excessively delaying packets from both flows due to backup and/or overflowing of queues." In particular the scenario which matches our requirement is the 'Y-topology' in which we are interested in detecting congestion between two flows originating at different sources (different servers/LPSS peers) and terminating at the same destination (requesting LPSS).

Therefore, a good algorithm for the peer selection process may proceed in two stages. The first stage will use Rubenstein et.al.'s shared congestion detection technique to cluster the set of servers and peers. All servers (peers) within a cluster have shared congestion, and it may be useful to choose only a subset, maybe one server (peer), from each cluster. The choice of the single server (peer) can be performed based on the bandwidth based heuristic, described earlier in this section. However, in our experiments on the PlanetLab, we found that in most cases, little shared congestion was detected between pairs of nodes. We believe this to be primarily due to the high bandwidth connectivity availed by most of such nodes as shown in [17]. Therefore, our PlanetLab experiments primarily tested the usefulness of the bandwidth-based heuristic.

In general, it is possible to be more efficient in the peer and server selection process by inferring more detailed characteristics of the network topology using network tomography approaches, as used in [10]. However, prior work [18] has shown that simple heuristics, similar to ours, are adequate for most practical purposes. Hence in our implementation we use our proposed approach and through experiments on PlanetLab we show its adequacy in wide-area deployments of peer-to-peer media streaming. Nevertheless, we believe that further investigations are necessary to evaluate and compare efficacy of lightweight approaches such as ours to more detailed network topology inferencing approaches as in [10] as well as diverse shared congestion detection and evaluation mechanisms as proposed in [13, 14, 15, 16]. We leave that as future work.

## 5. IMPLEMENTATION

We now proceed to describe a prototype implementation of the Π-Stream architecture. The Π-Stream system is implemented in C/C++. As described earlier, the key entity in the Π-Stream architecture is the LPSS. In order to decouple the media playback from the media download, the LPSS is implemented as two separate but interacting parts, the LPSS Downloader and the LPSS Streamer. The LPSS Downloader is implemented as a logical entity which maintains connections between servers, downloads the media blocks to the client's hard drive and runs the block scheduler when appropriate. The LPSS Streamer is a significantly modified version of Darwin Streaming Server 5.0.0.1 that streams the media from the client's hard drive to the player as the LPSS Downloader makes it available.

### 5.1 Execution

The media player requests a media object from its LPSS which in turn contacts the tracker. The tracker keeps records of what clients are currently downloading the file and what blocks of the file are currently available at each LPSS. Each LPSS is also able to request parts of the file from other clients which are downloading the file simultaneously. The block map of the file is available at each LPSS. The downloaded file is stored on the client hard drive until the download completes. HTTP byte-offset requests are used to retrieve data from the servers.

The block scheduler requests a block from each of the available servers/clients. Each second, the server's transfer rate during the current download is computed. (Rates when opening a new TCP connection and during the initial HTTP block request are not considered because they are poor representations of the actual server/client transfer rate) The instantaneous rate is used to update the server's transfer rate using an exponential averaging parameter of $\alpha = 0.8$. In order to identify the situation where a block is unlikely to be downloaded before its deadline, the client determines the amount of time required to download the remaining portion of the block at the current server transfer rate. If this time is larger than the block's playback deadline, the scheduler is informed of the possibility of a missed playout. The scheduler takes remedial action as described in the previous section.

If a block's playback deadline could not be met, the remainder of the block's data are marked as downloaded (and essentially ignored) and the download continues with the next sequential byte. This is done to prevent the block scheduler from attempting to create a feasible downloading schedule for a block that has already missed its deadline.

### 5.2 LPSS Torrent

Each LPSS begins communicating with the tracking server as it initiates a download. It reports what file it is currently downloading and how many bytes of the file it has downloaded. Subsequently, the client reports the amount of available data to the tracker after each block download is completed. Once a second, each LPSS queries the tracker for other clients downloading the same movie file. If such clients are available, they are added to the server list and the clients may retrieve blocks from each other as long as they are within the serving clients available byte range. All requests for blocks between LPSSs are once again HTTP byte-offset requests. To the LPSS, both other LPSS clients and the file servers appear the same, with the distinction that they may retrieve all blocks from the servers but only currently available blocks from the clients.

### 5.3 Optimizations

### *Choosing initial block sizes*

The size of downloaded blocks depends on the achievable transfer rate between the LPSS and its source. While such block sizes adapt quickly over time based on our defined split and swap algorithms, the initial choices of the block sizes can be quite bad. Instead of arbitrarily assigning a fixed size block to each source for the first download, we adopt a strategy whereby the LPSS tries to accumulate the largest possible amount of contiguous data within the initial buffering phase.

We explain our approach using a simple example. Consider 3 servers with a download rate of 200 KBps to the client and an initial buffering time of 5 seconds. During the buffering stage, each server is capable of downloading 1MB of data. However, if the blocks are 5 MB in size, the servers will download the 1st, 5th and 10th megabyte resulting in 1 MB of contiguous data when the movie begins streaming. If instead we use blocks 1 MB in size, the servers will download the 1st, 2nd and 3rd megabyte resulting in 3 MB of contiguous data after the same amount of buffering time. Decreasing the blocks in size further would result in more time being spent requesting blocks. Therefore, the ideal size of each assigned block during the buffering phase is equal to the rate of the server assigned to that block times the amount of buffering time, resulting in the amount of data downloaded equal to the sum of the server rates times the amount of buffering time. Since the buffering time is so short, we do not need to worry about serious server rate fluctuations. We employ this optimization in our implementation.

### *Managing TCP connections*

Since the Π-Stream system is built on TCP, the system suffers from the overhead of 3-way handshakes as well as TCP slow start. Since HTTP does not offer a mechanism for aborting ongoing transfers, each block split or swap resulting in a server reassignment before the download of the current block is completed results in the need to close and reopen the current connection between the server and client. This can have adverse effects on the transfer rate, especially if the RTT between the server and client is sufficiently large.

Therefore, to minimize the overhead of opening a new TCP connection, we instead maintain two TCP connections to each server or peer. The first (active) connection is used for most of the downloaded data, while the second (passive) connection is left open for failover when the first connection chokes during a split operation. If the LPSS executes a split or swap resulting in a block reassignment for the server, the data flowing through the current active connection is rendered useless. At that instant, the active connection is closed and the LPSS immediately begins to download new blocks on the previously passive, now active connection. Subsequently we open a new connection as a passive connection. We download a small fraction of data periodically through the passive connection to ensure that it does not stay in the TCP slow-start phase when it is converted to an active one. Note that such an approach effectively doubles the amount of TCP state maintained between a pair of nodes in the system but leads to improved transfer performance.

## 6. EXPERIMENTAL EVALUATION

**Table 1: A list of PlanetLab sites used for our experiments with Π-Stream.**

| Label | IP address | Domain name |
|---|---|---|
| A | removed for double | blind review |
| B | 143.89.126.12 | plab1.cs.ust.hk |
| C | 128.2.198.188 | planetlab-1.cmcl.cs.cmu.edu |
| D | 128.214.112.91 | planetlab1.hiit.fi |
| E | 140.112.107.80 | planetlab1.im.ntu.edu.tw |
| F | 142.103.2.1 | planetlab1.cs.ubc.ca |
| G | 158.130.6.254 | planetlab1.cis.upenn.edu |
| H | 128.112.139.71 | planetlab-1.cs.princeton.edu |
| I | 128.135.11.149 | planetlab1.cs.uchicago.edu |
| J | 171.64.64.216 | planetlab-1.stanford.edu |

We now evaluate the capabilities and limitations of the Π-Stream architecture. We do so using extensive testing of our prototype implementation on PlanetLab. We examine Π-Stream's ability to take advantage of available surplus bandwidth, its scalability and its resilience to the changing dynamics of a network. We first examine the performance of a single Π-Stream client and later extend our results to scenarios containing multiple clients.

We use multiple PlanetLab nodes to run our experiments. The nodes are listed in Table 1. Each node acting as a server uses Apache 2.0.54. Each client requesting a block from the server uses a HTTP byte-offset request corresponding to the first and last bytes of the requested block. This allows the clients to determine a file's block mapping without the need to communicate this mapping to the servers. We used MPEG4 content in the experiments and playback was provided to the user using MPlayer version v1.0pre6. Number of clients, playback rates, block mappings, and buffering times were varied in different experiments.

Prior to discussing these results, we should mention one peril of comparing these experiments to each other. In general, for an uncontrolled wide-area environment such as PlanetLab, the performance of each run is potentially independent of others, and heavily depends on the network conditions between the various PlanetLab nodes. However, the trends observed over a large set of our experiments show significant consistency and match our intuition of expected performance. We believe that such results indicate a reasonable stability in network conditions across quick sequence of experiments on the PlanetLab testbed.

Our results presented below illustrate Π-Stream performance for representative runs as well as aggregated results over multiple runs, as appropriate for each plot. In the data download throughput plots, the download throughput were averaged over individual one second intervals.

## 6.1 Block Scheduling

We first show some performance gains when streaming media using Π-Stream when operating in the LPSS-standalone mode. We use the nodes listed in Table 1. Our client machine is located in our laboratory (node A). The three servers are located at nodes B, C, and D (in Hong Kong, US, and Finland respectively). All of the servers contain the entire data file of size which is 50MB long. The targeted media playback rate is 2.4 Mbps and the buffering time is 6 seconds.

The first experiment examines the ability of the LPSS to

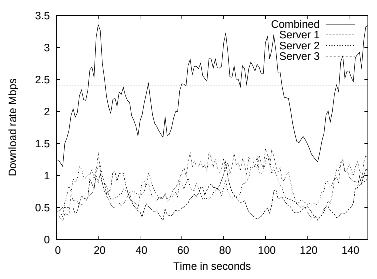

**Figure 5: Transfer rates between servers and the LPSS.**

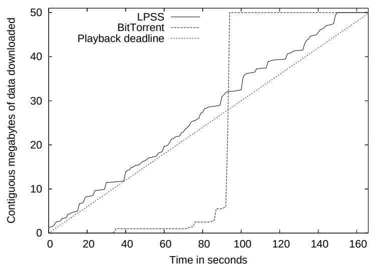

**Figure 6: Arrival before deadline using the LPSS block scheduler and a random block assignment scheduler.**

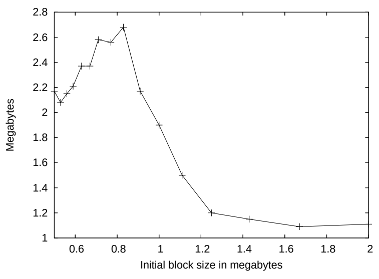

**Figure 7: Amount of initial contiguous data buffered during a 5 second buffering phase with varying block sizes.**

schedule blocks in a manner such that all blocks meet their playback deadlines.

In Figure 5, we plot the transfer rates between the LPSS and the three servers as they vary over time during the experiment. The horizontal line indicates the target playback rate of 2.4 Mbps. While the transfer rates from individual servers are lower than this playback rate, the aggregated transfer rate is higher than the targeted playback rate, on average. It is interesting to note that over the duration of experiment, the aggregate download rate often dips below the playback rate. However, the block scheduler, with its initially downloaded buffer window, is able to compensate for these fluctuations by performing appropriate splits and swaps, thus permitting the client to perceive zero data loss and a constant streaming rate of 2.4 Mbps throughout the course of the entire streaming process.

This can be better observed in Figure 6, where we plot the number of contiguous playback data that were downloaded over time. The diagonal line in the plot indicates the download deadline for the corresponding data for appropriate playback and the LPSS-standalone continuously meets the deadline. It is also interesting to compare the performance of the LPSS block scheduler against a BitTorrent style scheduler which does random block assignments for each server without regard for the block's playback deadlines (also shown in Figure 6). It is clearly illustrated that the LPSS block scheduler's understanding of block playback deadline, combined with its ability to split and swap blocks during downloads, results in the download of all blocks before their playback deadline. As expected, the BitTorrent scheduler's random block assignments prevent it from maintaining a good playback experience as many blocks miss their playback deadline.

## 6.2 Choosing block sizes

In order to improve initial performance of an LPSS, it is important to carefully choose the download block sizes for the first few blocks such that we can obtain the largest initial buffer window. An appropriate optimization was described in Section 5. We present an experimental evaluation of the proposed optimization in Figure 7. We use nodes B, C, and D as servers. For different runs, we use different initial block sizes (varied in the x-axis in the plot). It is easy to observe that the overheads of issuing small block download is high and the total contiguous data downloaded in the buffering phase increases with increase in block sizes, upto a block size of 0.83 MB. This is the best block size for the given scenario, where the chosen block sizes for the different servers leads to a perfect download of a contiguous set of data within the initial buffering phase — download completed by the first server is exactly contiguous to the download by the second server which is also exactly contiguous to the download by the third. However, increasing the initial block size to greater values leads to decreased performance due to gaps created between the data downloaded from each server in the buffering phase.

In Figure 8 we examine this further by examining the impact of adaptive block sizing on the number of blocks that are eventually split over the duration of the experiment. In the non-adaptive case, we statically chose each initial block to be 10 MB in length (which is a poor choice for this client and set of servers). The plot illustrates the number of times initially chosen blocks are split for each block in sequence. In the non-adaptive case, due to poor conservative choice of block sizes, initially assigned blocks are split more than 20 times to ensure they meet the playback deadline. In comparison, in the adaptive case, the initial block choices are performed more intelligently and hence leads to very few splits over the entire experiment and overall improved performance at the client.

**Table 2: Transfer rates of different peer LPSS nodes to A and their remaining time in the system when A joins for the results shown in Figure 9.**

| | LPSS peers | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | B | C | D | F | G | H | I | J |
| Transfer rate(Mbps) | 0.46 | 0.12 | 0.8 | 1.44 | 2.08 | 2.6 | 3.2 | 1.0 |
| Remaining Time(sec) | 80 | 70 | 30 | 40 | 10 | 50 | 20 | 60 |

## 6.3 Peer selection algorithm

We now examine the efficacy of the peer selection algorithm proposed in Section 4. In this experiment, we chose node A to be the client and node E to be a server. All the other nodes were LPSS peers that started before A. At the time A joins the system, different peers are in different stages of their own playbacks (see Table 2). In later experiments we will vary the number of LPSS peers a client is allowed to choose. In this experiment we allow A to choose upto 3 peer LPSS nodes for additional downloads. In the plot we illustrate the progress of data download over time with re-

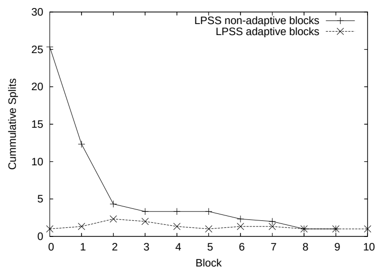

**Figure 8: Number of cumulative block splits for each block with and without communication between clients. In the adaptive case, only the first 10 blocks in sequence are shown.**

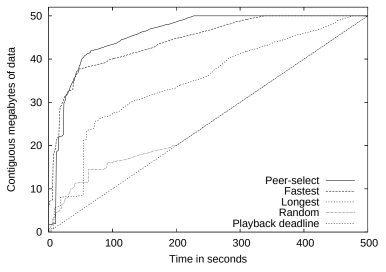

**Figure 9: Comparison of different possible peer selection techniques along with our proposed heuristic**

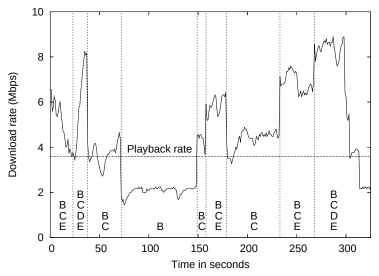

**Figure 10: LPSS throughput rate. (Dashed lines represent servers going up and server failures. Letters correspond to active servers in these periods.**

spect to the playback deadline for the data. Along with our proposed algorithm (marked as PeerSelect) we studied three other simple heuristic for peer selection: (i) random: where A picks three other LPSS peers at random, (ii) longest: where A picks 3 LPSS peers that are expected to stay the longest in the system, and (iii) fastest: where A picks the 3 LPSS peers with the highest transfer rate to itself. In contrast to these heuristics, our proposed algorithm selects a peer which can provide the most amount of useful data in its duration and hence leads to the most time efficient download. However, it is interesting to note that the longest and the fastest heuristics also provide reasonable performance in this scenario. Based on this and other experiments on peer selection (not reported) we concluded that a simple (and easy to implement) heuristic such as 'fastest' or our proposed 'PeerSelect' provides adequate performance.

## 6.4 Varying Network Conditions

We now examine Π-Stream's ability to handle unexpected data flow failures. For this experiment, we use the same client (node A) and nodes B, C, D, and E as the set of servers. We use a higher playback rate of 3.6 Mbps and a larger movie file of size 200 MB (since we increased the number of servers to four). We begin the movie download with three servers (B, C, and E). Over the period of this experiment, additional servers join while some existing servers fail (marked with dashed vertical lines in the figure).

While the download rate fluctuates with server failures and joins (Figure 10), the playback experience is unaffected at the client (Figure 11). Even though between time 72 and 149 seconds there is only one server in the system (server B), all downloaded data meet the playback deadline.

When node C finally becomes available at 149 seconds, the two servers are able to use their new combined bandwidth to replenish the depleted playback buffer and continue to do so as more servers become available. The ability to continuously leverage all available bandwidth allows Π-Stream to remain unaffected by future bandwidth fluctuations resulting from server failure and data congestion.

## 6.5 LPSS-Torrent

Now our analysis turns to the LPSS-Torrent scenario with multiple clients downloading the same file simultaneously. Nodes C, D and E act as the servers while nodes G, I and J act as the Π-Stream clients (these client nodes have lower access bandwidths than previously used client A). The targeted playback rate is 2.4 Mbps. In this LPSS-Torrent scenario, each client potentially downloads data from all servers as well as all other clients in the system. The Π-Stream clients are started 35 seconds apart beginning with G and ending with J. We analyze the performance gains of using multiple Π-Stream clients capable of requesting data from each other by comparing them to multiple Π-Stream clients that are unable to do so.

In Figures 12-14 we plot the download rates of each client when operating in the LPSS-Torrent and the LPSS-Standalone mode. In Figure 12 we notice little initial change between LPSS-Torrent and LPSS-Standalone. In the latter two figures, however, we can see an increasing difference in initial download rates. This is because of the ability of clients at nodes I and J to retrieve blocks from already active clients. For example, when the client at node I starts its download it can retrieve data from the 3 servers as well as whatever data client G currently has available. Similarly, the client at node J can initially retrieve data from the 3 servers and whatever data clients G and I have available at that time. The initial download rate of the client at node G remains unchanged between LPSS-Standalone and LPSS-Torrent. This is because in either case, this client is unable to find another client that is more advanced in the playback than itself.

In Figure 15 we plot the amount of data available for playback at each client node using LPSS-Torrent. Even though the clients begin their downloads 35 seconds apart, 130 seconds into the download their available playback data converges. This is a result of the fact that the clients at nodes I and J can take advantage of the extra data transfer between existing clients to catch-up in the download process. Since the client at node G has the most data, it cannot request data from other clients and continues to download at a more uniform rate. Once the amount of data available for playback converges for all servers, the clients are no longer able to request large blocks of data from each other. Since the clients are now forced to download almost all of their data from the 3 original servers, at this time the download rate of LPSS-Torrent becomes roughly equivalent to the download rate of LPSS-Standalone. This result can be clearly seen in Figures 12-14 as the download rate lines merge. As certain LPSS-Torrent clients regain a significant

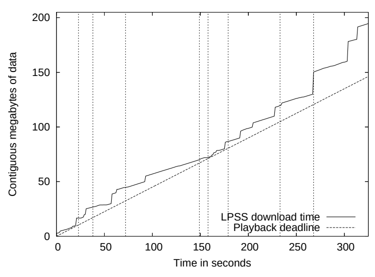

**Figure 11: Amount of data available for LPSS Streamer playback over the course of the download.**

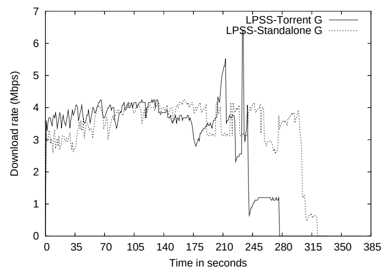

**Figure 12: Download rate of client G using LPSS-Torrent and LPSS-Standalone implementations**

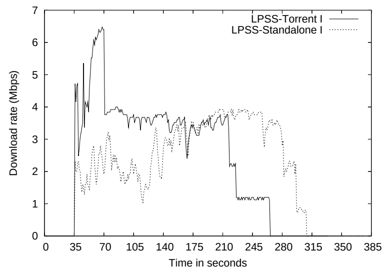

**Figure 13: Download rate of client I using LPSS-Torrent and LPSS-Standalone implementations.**

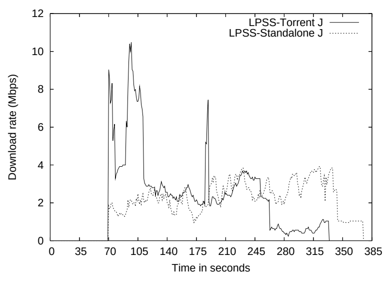

**Figure 14: Download rate of client J using LPSS-Torrent and LPSS-Standalone implementations.**

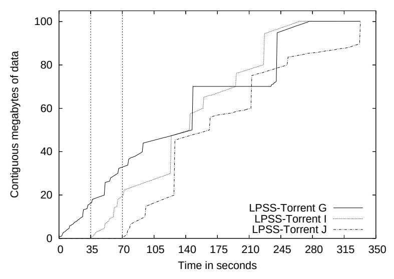

**Figure 15: Amount of data available for streaming at each Π-Stream client using LPSS-Torrent. Dashed vertical lines represent LPSS client joins.**

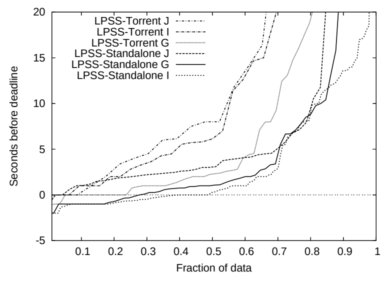

**Figure 16: Arrival times of data before playback deadlines for each Π-Stream client. Plot cutoff at 20 seconds before deadline.**

lead in downloaded data, we observe spikes in the download rates as large scale inter-client transfers occur once again. These spikes are most pronounced at 184 seconds for client J and at 210 and 234 seconds for client G.

In Figure 16 we plot the arrival time of data with respect to its playback deadline for all the clients in the two scenarios. It is easy to observe the overall improvement in data arrival time that LPSS-Torrent gives over LPSS-Standalone. Many of the blocks downloaded using LPSS-Standalone barely meet or miss the playback deadline (e.g., for LPSS-Torrent G, nearly 30% of its data misses the deadline), while the LPSS-Torrent clients are able to maintain the playback rate during the entire duration of the download. The client at node J shows the most improvement, followed by the client at node I and finally the client at node G. The client at node J shows the most significant improvement because it is the last to join the file download and can take advantage of the additional data transfer from the first two clients. Similarly, the client at node G shows little improvement due to the fact that it almost always has the largest amount of data and cannot request blocks from the other clients. Therefore, in both LPSS-Torrent and LPSS-Standalone, the client at node G downloads almost all of its data from the 3 servers at the same download rate.

The main advantage of the LPSS-Torrent is that it eases the load at servers by distributing requests between Π-Stream peers as well as the servers.

## 6.6 Varying number of servers and LPSS peers

We now examine Π-Stream's inherent ability for scalability, first for LPSS-Standalone and then for LPSS-Torrent. Our client machine is located in our laboratory (Node A). In the LPSS-Standalone experiment, we varied the number of available servers from 1 to 9 to download a 200 MB media file for real-time playback. For each choice of number of servers, we included all servers in the previous set plus one additional server in the order listed in Table 1. Essentially, the servers are added from the top to the bottom of the table, beginning with node B. The maximum sustainable streaming rate is defined as the maximum media playback rate the system can support throughout the entire download with no data loss.

In Figure 17 we show this maximum sustainable streaming rate that is possible as the number of servers hosting the file increases. With one server, the maximum sustainable streaming rate is limited by the download rate between the LPSS and the server. With two or more servers, the maximum sustainable streaming rate is greater than either of the sole server's maximum rate due to the fact that blocks can be split or swapped between servers to increase the *overall* download rate. In single client-server data shown on the same plot, we picked only one server out of the set of corresponding servers which had the highest bandwidth to the client. Clearly in this non-LPSS scenario, the sustainable streaming rate would be limited by the connection rate between the client and the fastest server.

While in this experiment it turned out that the maximum sustainable streaming rate monotonically increases with increase in the number of servers, clearly such a trend cannot continue forever, and will be eventually limited by the access

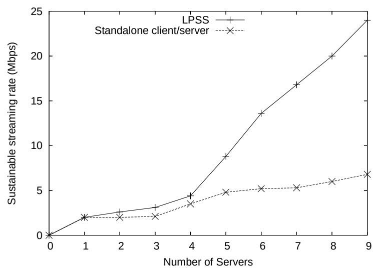

**Figure 17: Maximum sustainable streaming rate per number of servers.**

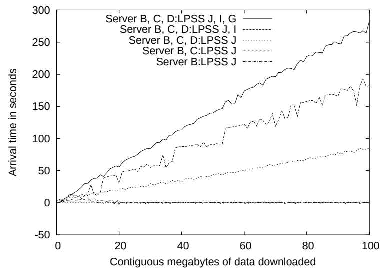

**Figure 18: Arrival of data with respect to playback deadline over multiple configurations of servers and LPSS clients.**

bottleneck of the client. Nevertheless, the experiment confirms the natural intuition that in many realistic settings, a client can significantly improve its streaming performance by adding multiple distributed servers.

Next we examine the performance of Π-Stream operating in LPSS-Torrent mode as we increased the number of servers as well as the number of LPSS peers. We use nodes B, C and D as the servers and nodes G, I and J as the LPSS clients. The movie file is 100 MB in size while the targeted playback rate is 2.4 Mbps. In Figure 18 we plot the arrival time of each byte before its deadline using various server/client configurations. If the time is positive, the byte arrived before the playback deadline, if the time is negative the byte arrived after its playback deadline and if the time is zero the byte arrived right at the playback deadline.

We first use G as the client and vary the number of servers. The one and two server configurations cannot support the playback rate and after the initial buffering phase, the arrival times quickly dip below the playback deadline The arrival times on the graph never miss the deadline by more than a few seconds because the block is marked as done as soon as the the playback deadline is missed and download of the next block immediately begins. Loss rates are higher than 50% in both cases, once the blocks begin missing playback deadlines. Using three servers, the LPSS can sustain the streaming rate and the arrival times remain positive throughout the download.

Next we keep the three servers and vary the number of LPSS clients. We start each client 50 seconds apart and only plot the results for the last LPSS client to join since it experiences the greatest performance gain. As more LPSS clients download the movie, the joining clients can take advantage of the data downloaded by the existing clients to fetch blocks sooner, therefore increase the arrival time of the block before its playback deadline. In Figure 18, we can clearly seen an increase in the arrival times with multiple existing clients.

## 6.7 Result Summary

We have shown that our implementation fulfills many of the goals described in our Π-Stream architecture. It has the ability to scale to an arbitrary number of servers and use all available bandwidth. It has the ability to compensate for future bandwidth fluctuation by using a large download buffer window. It allows for stable, long-term high-quality playback regardless of unexpected server failure. It reduces server load by requesting blocks at the highest possible granularity. It has the ability to use block splitting and block swapping in order to increase the *overall* download rate.

## 7. RELATED WORK

There is a large body of work that is closely related to Π-Stream. P2P style media streaming is not a new concept and has been extensively researched over the last few years. A number of research projects have examined the problem of one-to-many (multicast) of media streams using an organized application-layer overlay structure, e.g., End System Multicast [19], NICE [20], CoopNet [21]. The focus of these approaches is in efficient construction of a single or multiple trees for delivery of media streams through the client overlay. In Π-Stream we focus more on the complementary problem — how to efficiently stream data mirrored or stored at multiple locations to a single client and enhance its performance through P2P style interactions.

In P2Cast [22], the authors propose a similar overlay construction technique for a video-on-demand service. The emphasis of their work is on on “patching” by which a late joining client retrieves the initial missing portion of the stream from a server. The Cache-and-Relay media streaming work also examines such a patching question for delivery of such streams to a large number of asynchronous clients [23]. PROP [24] proposes the use of a structured overlay (e.g., Chord or CAN) over which stream segments are similar cached and routed to interested clients. The layered P2P streaming work considers a layer-encoded media streaming system in the P2P context [25]. In this work, the authors examine how each peer may find other peers from which to obtain a relevant set of layers for itself, given its bandwidth constraints, thereby maximizing a aggregate quality metric.

PROMISE proposes another P2P architecture for media streaming in which the media file is resident in multiple peers. The key objectives of PROMISE is to identify a good set of peers, using an overlay substrate such as Pastry [26], and receive the media stream from a subset of such peers.

More recently DONET/CoolStreaming proposed another overlay-based P2P media streaming solution (though specifically for live streaming as opposed to stored media streaming) where the peers maintain some partnerships with each other [27]. Each such peer instead of obtaining the media stream directly from an origin server, acquires it from its partner peers. Current versions of DONET are implemented for RealPlayer and Windows Media Player.

Finally, as described in Section 1, PALS defines a receiver-driven approach for P2P media streaming [9].

Our proposed Π-Stream system differs from all of the above in the following fundamental ways.

- Π-Stream advocates an indirect streaming approach in which the LPSS is a separate entity located in the client's machine and is responsible for downloading the media content from multiple servers and other peers.
- Π-Stream decouples the media download process from that of media streaming. As a consequence, it seamlessly integrates with any publicly available media streaming client such as Windows Media Player, QuickTime player, and RealPlayer, without requiring any change or update in the latter. In fact, the implementation of each of these players restrict them to connect to a single streaming server only. Π-Stream, through indirect streaming, allows such players to take advantage of availability of multiple servers as well as other peers interested in the same media content.
- As a consequence of such a design, Π-Stream is media-transparent and can be employed to stream any media format. Such media streaming can be enabled by placing the content with widely deployed web servers and FTP servers without even requiring deployment of expensive streaming servers in the Internet.

## 8. CONCLUSION

We designed Π-Stream, an indirect streaming architecture for streaming media to a client from multiple sources, both servers and peers. Indirect streaming decouples the media playback from the process of media downloading and has numerous advantages. We have implemented our Π-Stream architecture and conducted extensive tests on the wide-area PlanetLab testbed. Our results indicate that such an approach is viable for sustaining high data rate playback (more than 2.4 Mbps) in various wide-area scenarios.

## 9. REFERENCES

[1] "http://www.nielsen-netratings.com/pr/pr_04081."

[2] S. S. A. Akella and A. Shaikh, "An empirical evaluation of wide-area internet bottlenecks," in *Internet Measurement Conference*, 2003.

[3] A. K. P. Rodriguez and E. W. Biersack, "Parallel-access mirror sites in the internet," in *IEEE Infocom*, Mar. 2000.

[4] M. L. J.W. Byers and M. Mitzenmacher, "Accessing multiple mirror sites in parrallel: Using tornado codes to speed up downloads," in *IEEE Infocom*, 1999.

[5] "Bit torrent, http://www.bittorrent.org."

[6] "E-mule project, http://www.emule-project.net/."

[7] H. Schulzrinne, G. Gokus, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," *RFC 1889*, Jan. 1996.

[8] H. Schulzrinne, A. Rao, and R. Lanphier, "Real time streaming protocol: RTSP," *RFC 2326*, Apr. 1998.

[9] R. Rejaie and A. Ortega, "Pals: Peer-to-peer adaptive layered streaming," in *NOSSDAV*, 2003.

[10] M. Hafeeda, B. Habib, A. Botev, D. Xu, and B. Bhargava, "Promise: Peer-to-peer media streaming using collectcast," in *Proceedings of ACM Multimedia*, 2003.

[11] R. Rejaie, H. M., and D. Estrin, "Rap: An end-to-end rate-based congestion control mechanism for realtime streams in the internet," in *IEEE Infocom*, 1999.

[12] J. P. S. Floyd, M. Handley and J. Widmer, "Equation-based congestion control for unicast applications," in *ACM Sigcomm*, 2000.

[13] D. Rubenstein, J. Kurose, and D. Towsley, "Detecting shared congestion of flows via end-to-end measurement," *. IEEE/ACM Transactions on Networking*, vol. 10, no. 3, June 2002.

[14] D. Katabi, I. Bazzi, and X. Yang, "A passive approach for detecting shared bottlenecks," in *International Conference on Computer Communications and Networks*, Oct. 2001.

[15] K. Harfoush, A. Bestavros, and J. Byers, "Robust identification of shared losses using end-to-end unicast probe," in *International Conference on Network Protocols*, Nov. 2000.

[16] M. Kim, T. Kim, Y. Shin, S. Lam, and E. Powers, "A wavelet-based approach to detect shared congestion," in *ACM Sigcomm*, Aug. 2004.

[17] S. Banerjee, T. Griffin, and M. Pias, "The interdomain connectivity of planetlab nodes," in *Passive and Active Measurements workshop*, 2004.

[18] K. Hanna, N. Natarajan, and B. Levine, "Evaluation of a novel two-step server selection metric," in *International Conference on Network Protocols*, Nov. 2001.

[19] Y.-H. Chu, S. G. Rao, and H. Zhang, "A case for end system multicast," in *ACM Sigmetrics*, 2000.

[20] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *ACM Sigcomm*, 2002.

[21] P. C. V. Padmanabhan, H. Wang and K. Sripandikulchai, "Distributing streaming media content using cooperative networking," in *NOSSDAV*, May 2002.

[22] J. K. Y. Guo K. Suh and D. Towsley, "P2cast: Peer-to-peer patching scheme for vod service," in *Proc. WWW'03*, May 2003.

[23] S. Jin and A. Bestavros, "Cache-and-relay streaming media delivery for asynchronous clients," in *International Workshop on Networked Group Communication NGC'02*, Oct. 2002.

[24] X. C. L. Guo, S. Chen and S. Jiang, "Prop: a scalable and reliable p2p assisted proxy streaming system," in *ICDCS*, Mar. 2004.

[25] Y. Cui and K. Nahrstedt, "Layered peer-to-peer streaming," in *NOSSDAV*, June 2003.

[26] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001.

[27] B. L. X. Zhang, J. Liu and T.-S. Yum, "Coolstreaming/donet: A data-driven overlay network for live media streaming," in *IEEE Infocom*, Mar. 2005.