On-Demand Media Streaming over the Internet*

Mohamed M. Hefeeda and Bharat K. Bhargava
CERIAS and Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{mhefeeda, bb}@cs.purdue.edu

Abstract

We propose a new model for on-demand media streaming centered around the peer-to-peer (P2P) paradigm. The proposed P2P model can support a large number of clients with a low overall system cost. The P2P model allows for peers to *share* some of their resources with the system and in return, they get some *incentives* or rewards. We describe how to realize (or deploy) the proposed model. In addition, we present a new *dispersion* algorithm (for disseminating the media files into the system) and a *searching* algorithm (for locating peers with the required objects).

We demonstrate the potential of the P2P model as an infrastructure for a large-scale on-demand media streaming service through an extensive simulation study on large, Internet-like, topologies. Starting with a limited streaming capacity (hence, low cost), the simulation shows that the capacity is rapidly increased and many clients can be served even if they come according to different arrival patterns such as constant rate arrivals, flash crowd arrivals, and Poisson arrivals.

1 Introduction

Streaming multimedia files to a large number of customers imposes a high load on the underlying network and the streaming server. The voluminous nature of the multimedia traffic along with its timing constraints make deploying a large-scale, cost effective, media streaming architecture over the current Internet a challenge.

The current media streaming architectures are mainly composed of a streaming entity and a set of requesting clients. The supplying entity could be one server, a set of servers, a set of servers and caches, or a set of servers and proxies. This entity is responsible for providing the requested media files to *all* clients. Figure 1 depicts a typical streaming architecture. The total number of concurrent clients the system can support, called the overall system capacity, is limited by the resources of the streaming entity. The limitation mainly comes from the out bound network bandwidth, but it could also be due to the processing power, memory size, or the I/O speed of the server machine. For instance, a streaming server hooked to the Internet through a T3 link ($\sim$ 45 Mb/s) would be able to support up to 45 concurrent users requesting constant bit rate (CBR) media files recorded at 1 Mb/s. These approaches have limitations in reliability and scalability. The reliability concern arises from the fact that only one entity is feeding all clients, i.e., a single point of failure. The scalability of these approaches is not on a par with the requirements of a media distribution service that spans Internet-scale potential users, since adding more users requires adding a commensurate amount of resources to the supplying server.

Whereas deploying proxies and caches at several locations over the Internet increases the overall system capacity, it multiplies the overall system cost and introduces many administrative challenges such as cache consistency and load balancing problems. The system's overall capacity is still limited by the aggregate resources of the caches and proxies. This shifts the bottleneck from one central point to a "few" distributed points, but it does not it.

We propose a peer-to-peer media distribution model that can support a large number of clients with a low overall system cost. The key idea of the model is that peers *share* some of their resources with the system. In return, they get some *incentives* or rewards from the service provider. As peers contribute resources to the system, the overall system capacity increases and more clients can be served. By properly motivating peers, the service provider can achieve a large system capacity with a relatively small initial investment. A peer-to-peer architecture has the potential to provide the desired large-scale media distribution service. The success of peer-to-peer file sharing systems such as Gnutella [18] and Napster [25]. show a proof of concept. The proposed architecture takes peer-to-peer file sharing systems a step further to provide a global media distribution service.

*This research is sponsored in part by the National Science Foundation grants CCR-991712 and CCR-001788, CERIAS, and IBM SUR grant.

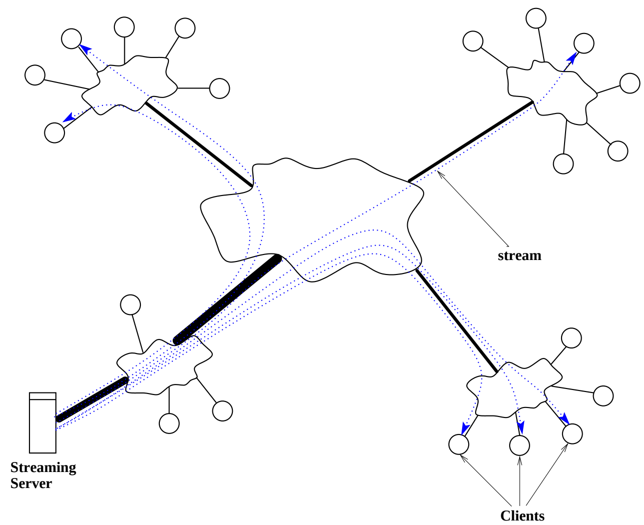


Figure 1: Abstract view of the current media streaming architectures. All clients are served by the streaming server.

There is a difference between a file-sharing system and a media streaming system [27]. In file-sharing systems, a client first downloads the *entire* file before using it. The shared files are typically small (few Mbytes) and take a relatively short time to download. A file is stored entirely by one peer and hence, a requesting peer needs to establish only one connection to download it. There are no timing constraints on downloading the fragments of the file, rather the total download time is more important. This means that the system can tolerate inter-packet delays. In media streaming systems, a client *overlaps* downloading with the consumption of the file. It uses one part while downloading another to be used in the immediate future. The files are large (on the order of Gbytes) and take long time to stream. A large media file is expected to be stored by several peers, which requires the requesting peer to manage several connections concurrently. Finally, timing constraints are crucial to the streaming service, since a packet arriving after its scheduled play back time is useless and considered lost.

The main contributions of this paper can be summarized as follows. We propose a new P2P media streaming model, which is suitable for an on-demand media distribution service. Second, we propose a new P2P streaming protocol to be used by a participating peer in the system. Third, we describe an architecture to realize (or deploy) the proposed model. For this architecture, we propose new *dispersion* and *searching* algorithms. The dispersion algorithms efficiently disseminate the newly published files into the system. The searching algorithms are for locating peers with the required objects. Fourth, we demonstrate the potential of the P2P model as an infrastructure for a large-scale on-demand media streaming service through a simulation study on large, Internet-like, topologies. We evaluate several performance measures of the proposed model under different client arrival patterns such as constant rate arrivals, flash crowd arrivals, and Poisson arrivals.

The remainder of the paper is organized as follows. Section 2 presents the P2P model. Section 3 presents the protocol to be run by a participating peer in the system. The architecture along with the searching and dispersion algorithms are presented in Section 4. The simulation study is presented in Section 5. Section 6 summarizes the related research effort. Section 7 concludes the paper and proposes future extensions for this research.

2 P2P Model for Media Streaming

The basic idea of our approach is shown in Figure 2. In the P2P model, a peer may act as a client and/or as a mini-server. As a client, it requests media files from the system. A peer may opt to store segments of the media files that it has already consumed for a specific period of time. As a mini-server, it can provide these segments to other requesting peers in the system. We emphasize the miniature attribute of the mini-server, since the peer was never intended to function as a full server. Instead, it serves a few peers for a limited duration. Each of these mini-servers adds only a little to the overall system capacity. Combining a large number of them can significantly amplify the capacity of the system. Peers join the system along with their resources. More cooperating peers results in an increase in the system capacity. This leads to a scalable system that can potentially support an enormous number of clients.

The system as a whole benefits from the cooperative peers. A well designed peer-to-peer system should provide

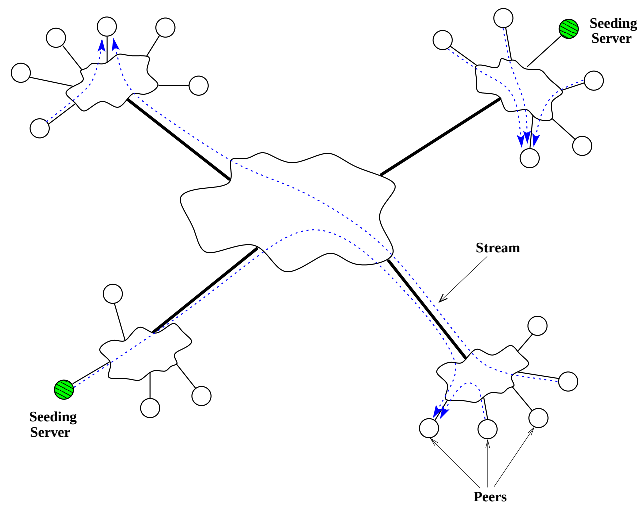


Figure 2: The proposed P2P architecture for media streaming. Peers help each other in streaming the requested media files.

sufficient *incentives* to motivate peers to share their storage capacity as well as their network bandwidth. In a recent study of two popular peer-to-peer file sharing systems (Napster and Gnutella), Saroui *et al.* [24] discovered that peers tend to avoid sharing their resources without enough incentives. The incentives may include lower rates ($/Byte) for those who store and supply media files to other peers in the system. Another way to encourage peers to share their resources is the "rewards for sharing" mechanism [6]. By this mechanism, points or credits are given to a cooperative peer as it increases the sharing. Consuming peers, get penalized by paying more to get resources from the system. Our cost-profit analysis considers incentives to the cooperative peers and studies how these incentives affect the profit of the service provider [7].

2.1 The Model

As shown in Figure 2, the model consists of a set of peers. We have a set of seeding servers that provide or *seed* the newly published media files into the system. They stream these files to a limited number of peers, which in turn, will feed another set of peers. After a short period of time, the system will have sufficient peers that already have the newly published media to satisfy almost all requests for the file without having to overload the seeding servers. We formally define the entities involved in our model as well as their roles and how they interact with each other in the following.

1. **Peers.** This is a set of nodes currently participating in the system. Typically, these are machines of the clients who are interested in some of the media files offered by a streaming center. Let $\mathbb{P} = \{P_1, P_2, \cdots, P_{\mathcal{N}}\}$ be the set of all peers in the system. Every peer $P_i, 1 \leq i \leq \mathcal{N}$, specifies three parameters: (1) R_i (in Kb/s), the maximum rate peer P_i is willing to share with others; (2) G_i (in bytes), the maximum storage space the peer is willing to allocate to store segments of one or more media files; and (3) C_i, the maximum number of concurrent connections that can be opened to serve requesting peers. A peer is not meant to be a server, since it has limited resources. By using these three parameters, a peer has the ability to control its level of cooperation with other peers in the system.

2. **Seeding server.** One of the peers or a subset of them may seed the new media files into the system. In a *commercial* media streaming service, these seeding servers will typically be owned by a service provider. For the abstract model, it does not make any difference whether the seeding server is just a peer or a dedicated server. In contrast, the realization of the model and the deployable architecture may differ if a dedicated server exists. Because this server may also be used to facilitate some crucial functions such as searching, dispersion, and accounting.

   We chose the name *seeding* (not streaming) servers to indicate that their main functionality is to *initiate* the streaming service and not to serve all clients at all times. These seeding "servers" do not negate the P2P nature of the model, in which peers help each other in streaming the media files.

3. **Stream.** A stream is a time-ordered sequence of packets belonging to a specific media file. This sequence of packets is not necessarily downloaded from the same serving node. Neither is it required to be downloaded in order. It must be displayed by the client in a specific order. It is the responsibility of the *scheduler* to download the packets from a set of possible nodes before their scheduled display time to guarantee non disruptive playing of the media.

4. **Media files.** The set of movies currently available in the system or offered by the media center. Let $\mathbb{M} = \{M_1, M_2, \cdots, M_m\}$ be the set of all available movies in the system. Every movie has a size in bytes, and is recorded at a specific bit rate R Kb/s. We assume that R is a constant bit rate (CBR). A media file is divided into N segments. A segment is the minimum unit which a peer can cache. A supplying peer may provide the cached copy of the segment at a rate lower than the required rate R. In general, one segment can be streamed to the requesting peer from multiple peers at the same time. According to our protocol (see Section 3), every peer will supply a different *piece* of the segment proportional to its streaming rate.

3 P2P Streaming Protocol

We describe the building blocks of the protocol used by a participating peer in the system. As shown in Figure 4, the protocol is composed of three phases and is to be run by a peer requesting a specific media file. In phase I, the requesting peer checks for the availability of the desired media file in the system. The phase starts with a crucial *searching* step. We describe the searching technique in Section 4.

The information returned by the searching step is arranged into a two-dimensional table. Each row j of the table contains all peers that are currently caching segment s_j of the requested file. Certain information about each peer is stored; e.g., its IP address, the available streaming rate, and some reliability information from the peer's history. Each row is then sorted to select the most suitable peers to stream from. Several criteria can be used for sorting, such as proximity to the client (in terms of network hops), available streaming rate, and peer's average on-line time. A weighted sum of some (or all) criteria could also be used. In our experiments, we use the proximity as the sorting criterion. This reduces the load on the network, since traffic will traverse fewer domains. In addition, the delay is expected to be shorter and less variable, i.e., smaller jitter. Phase I ends with a verification step to make sure that all segments are available either solely from other peers or from peers and seeding servers together. Otherwise, the requesting client backs off and tries later after exponentially increasing the waiting time.

The streaming phase starts only if phase I successfully finds all segments. Phase II streams segment by segment. It overlaps the streaming of one segment with the consumption of the previous segment. The playback of the media file starts right after getting the first segment. Because of the variability in network and peer conditions, buffering few segments ahead would result in a better playback of the media file. The buffering time can hide transient extra delays in packet arrivals. In case that one of the supplying peers fails or goes off line, this buffering time may hide delays due to finding and connecting to another peer from the standby table.

For every segment s_j, the protocol concurrently connects to all peers that are scheduled to provide pieces of that segment. The connections remain alive for time δ, which is the time to stream the whole segment. Different non-overlapping pieces of the segment are brought from different peers and put together after they all arrive. The size of each piece is proportional to the rate of its supplying peer. Let us define $\mathbb{P}^j$ as the set of peers supplying segment j. If a peer $P_x \in \mathbb{P}^j$ has a rate $R_x \leq R$, it will provide $|s_j|(R_x/R)$ bytes starting at wherever peer P_{x-1} ends. Since every peer supplies a different piece of the segment and $\sum_{x=1}^{|\mathbb{P}^j|} |s_j|(R_x/R) \geq |s_j|$, all pieces of the segment will be downloaded by the end of the δ period. To illustrate, Figure 3 shows three peers P_1, P_2, P_3 with rates $R/4, R/2, R/4$, respectively. The three peers are simultaneously serving different pieces of the same segment (of size 1024 bytes) to peer P_4.

Finally, in phase III, the peer may be allowed to cache some segments. This depends on the dispersion algorithm used. We present dipersion algorithms in Section 4.

4 Architecture

Two approaches may be used to realize the P2P streaming service model. The first approach relies on having a special entity to maintain information about the currently participating peers. We call it the *index* approach. If the seeding entity is a set of servers owned by a provider, the index will typically be maintained by this set of servers. The details of this approach are in the following subsections. The second approach does not assign special roles to any peer. It needs to logically interconnect peers in the system, we call this the *overlay* approach. We are currently working out the

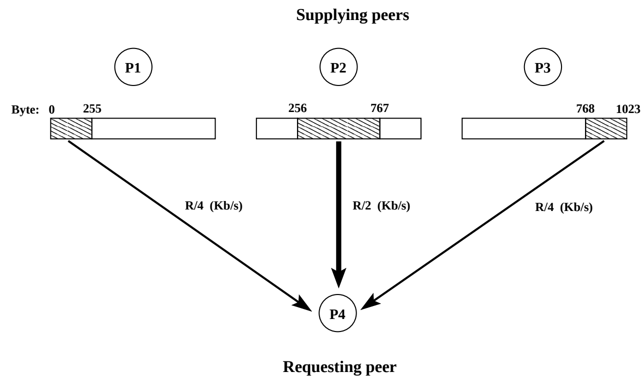


Figure 3: Peers P_1, P_2, and P_3 serving different pieces of the same segment to peer P_4 with different rates.

Protocol P2PStream

Phase I: Build AvailabilityTable (who has what)

a. Search for peers that have segments of the requested media file
b. Arrange the collected data in a two-dimensional table
/* Row j contains the set of peers willing to provide segment s_j */
let $\mathbb{P}^j = \{P_x | P_x \in \mathbb{P}$ and P_x is listed in row j $\}$
c. Sort every row of the table in ascending order /* Based on the proximity */
d. Verify the availability of all segments with the full rate as follows:
for $j = 1$ **to** N **do**
  if $\sum_{P_x \in \mathbb{P}^j} R_x \geq R$ **then**
    /* All pieces of segment s_j are available */
    Choose "sufficient" peers to provide the required R
    Put the rest in a "stand by" table. /* might be used if a peer fails */
  else if one of the seeding servers can provide the deficit for s_j **then**
    Add one more entry to row j with $(seedServerID, R - \sum_{P_x \in \mathbb{P}} R_x)$
  else /* all seeding servers are busy */
    Back off, wait expo increasing time after every failed trial
  end if
end for

Phase II: Streaming

let δ = time to stream a segment
let $t_1 = 0$
let $t_j = t_{j-1} + \delta, j \geq 2$
for $j = 1$ **to** N **do**
  At time t_j, get segment s_j as follows:
  let $b_0 = 0$
  for all $P_x \in \mathbb{P}^j, 1 \leq x \leq |\mathbb{P}^j|$ **do** /* in parallel */
    Connect to P_x
    let $b_x = |s_j| \frac{R_x}{R}$
    Download the piece of segment s_j from byte b_{x-1} to byte $b_{x-1} + b_x - 1$
  end for
end for

Phase III: Caching

Store some segments (determined by the dispersion algorithm and the peer's level of cooperation)

Figure 4: The protocol used by a peer requesting a media file.

details of this approach [8]. Both approaches follow the P2P paradigm, in which peers help each other in providing the *streaming* service. The two approaches are different in handling the *preparatory* steps of the streaming phase. The most important of these steps are: locating peers with the required media file (*searching*), and quickly disseminating media files into the system (*dispersion*). The chief distinction stems from the existence and the role of the *seeding* entity.

Before we present the index approach, we describe the *client clustering* idea, which is a key issue in the architecture. A cluster is defined as a logical grouping of clients that are topologically close to each other and likely to be within the same network domain [10]. It is highly beneficial for both the client and the network if a request can be fullfiled by peers within the same domain. For the network, it means that the traffic will travel fewer hops and hence will impose less load on the backbone links. The traffic delay will be shorter and less variable within the same domain, which is a desirable property for the streaming service.

We use a client clustering technique similar to the one proposed in [10]. The technique uses routing tables gathered from several core BGP routers. Client IP addresses that have the same longest prefix match with one of the routing table entries are assigned the same cluster ID. To illustrate the idea, consider five peers $P_1, P_2, P_3, P_4,$ and P_5, with IP addresses 128.10.3.60, 128.10.3.100, 128.10.7.22, 128.2.10.1 and 128.2.11.43, respectively. Suppose that among many entries in the routing tables, we have the following two entries: 128.10.0.0/16 and 128.2.0.0/16. The first three peers (all within Purdue University) share the same prefix of length 16 with the entry 128.10.0.0/16 (Purdue domain) and a prefix of length 12 with the entry 128.2.0.0/16 (CMU domain). Therefore, peers $P_1, P_2,$ and P_3 will be grouped together in one cluster with ID 128.10.0.0/16. Similarly, peers P_4 and P_5 will be grouped together in another cluster with ID 128.2.0.0/16. Notice that, using the same idea, a finer clustering within the same domain is also possible. For instance, P_1 and P_2 may be grouped in a smaller cluster with ID 128.10.3.0/24. This clustering technique does not incure much overhead, since it is performed once when the peer first joins the system.

4.1 Index Approach

Similar to Napster [25], the index approach requires one (or a small subset) of the participants to maintain an *index* to all other peers in the system. The index can be maintained by the same machine seeding the media files (i.e., the seeding server), or by a separate machine. In any case, we call the maintainer of the index as the index server. This approach may be described as a *hybrid* scheme because the streaming process is peer-to-peer, while the searching and the dispersion processes are server-assisted. The main role of this special node is *not* to provide the streaming service, but to facilitate the searching and the dispersion processes. The load, in terms of CPU, bandwidth, and storage, imposed by the control information required by the searching and dispersion processes is a small fraction of the load imposed by the streaming service. To some extent, this alleviates the scalability and the single point of failure concerns that typically arise in such architectures. This approach greatly simplifies the searching process and reduces the overhead associated with it. Without the index, the overhead traffic puts a non-negligible load on the system. The index approach is *practically* easier and faster to deploy and more appropriate for a commercial media provider, since a commercial media provider would keep a server for accounting and charging customers and to *seed* the newly available media files into the system.

4.1.1 Index Searching

A key issue in the index approach is to keep the index current. First, notice that peers who are currently caching some of the media files are known to the index. Because they initially contact the index server to get served those media files. And, it is the index server that decides for them what to cache, as explained in the next subsection. Therefor, the index already knows who has what. The index server, though, does not know whether a peer is currently on or off line. Several techniques may be employed to keep the index up to date. In the case that a peer gracefully shutts down, a daemon running on the peer can send a notification message to the index server. Since it is unlikely that too many peers shut down synchronously, these notification messages will not cause message implosion at the index server. Another way to keep the index server current is to have the requesting client checks the list of candidate peers returned by the index server by, for example, pinging them. The client then reports to the index server the status of all peers in the candidate list in one message.

The searching process is greatly simplified by the index server because it has a global information about all peers in the system. Figure 5 summarizes the searching process in the index approach. We assume that the index server gets the BGP routing tables and builds the clustering database apriori. Upon receiving a query from a client asking for a specific file, the index server first identifies the cluster to which the client belongs. If peers within the same cluster can satisfy the request, those peers will be returned to the client as a set of candiates to stream the request. Otherwise, peers from the closest clusters are chosen to serve the request. To find the closest clusters in terms of network hops, the same clustering idea can be applied *recursively*, that is, several smaller clusters are grouped together into a larger cluster if

Algorithm IndexSearch

/* Index server: upon receiving a query from peer P_r */
c $\leftarrow getCluster(P_r)$
for $j = 1$ **to** N **do** /* for every segment in the file */
  $candList[j] \leftarrow$ peers in c that have segment s_j
  if $\sum_{P_x \in candList[j]} R_x < R$ **then**
    if Peers from other clusters can provide the shortage **then**
      Append to $candList[j]$ sufficient peers from the *closest* clusters
    else
      return empty list to P_r /* P_r backs off */
    end if
  end if
end for
return $candList$ to P_r

Figure 5: Index-based Searching algorithm

Table 1: Symbols used in the `IndexDisperse` algorithm.

Scope	Symbol	Description
System	A	Average number of copies of the movie cached by all peers in the system
Variables	Q	Average movie request rate in the system
Cluster	L_c	Next segment to cache in cluster c
Variables	a_c	Average number of copies of the movie cached by peers in cluster c
	q_c	Movie request rate in cluster c
Peer	N_x	Number of segments cached by peer P_x
Variables	R_x	Rate at which peer P_x streams
	u_x	Fraction of time peer P_x is online
Movie	N	Number of segments of the movie
Variables	T	Duration of the movie (in hours)
	R	Rate at which the movie is recorded (CBR)

they share the same common network prefix. The index server, then, tries to satisfy the client's request from the larger cluster. For example, if we have peers P_1, P_2, P_3, P_4, and P_5, as described above, and P_1 is requesting a file. The index server will first try to satisfy the request from peers located within the cluster with ID 128.10.3.0/24, i.e., from peer P_2. If P_2 can not fullfill the request, the index server will try peers within the larger cluster with ID 128.10.0.0/16, i.e., from peers P_2 and P_3. If P_2 and P_3 can not fullfil the request, the index server will try to find peers from other clusters to make up the shortage. If the request can be fullfilled by any set of peers, this set is returned to the requesting client as a list of candidate peers. If the system does not have sufficient peers to satsify the request, an empty candidate peers list is sent to the client. The client then backs off and tries after an exponentially increased waiting time.

4.1.2 Index Dispersion

Caching the right segments of the media file at the right places is crucial to the incremental expansion of the system's capacity. The objective of the dispersion algorithm is to store enough copies of the media files in each cluster to serve all expected client requests from that cluster. As described in [7], peers may need some *incentives* to cooperate; especially, if the service is provided by a commercial provider. These incentives are costs imposed on the provider. For this reason, it is important to keep just the required capacity in the system. To do so, we propose a dynamic dispersion algorithm that adjusts the capacity within each cluster according to the average number of client requests from that cluster.

The dispersion algorithm works in the following setting. At a specific instant of time, the system can serve a certain number of requests concurrently. A client P_y sends a request to the system to get the media file. The client also declares its willingness to cache up to N_y segments to serve them to other clients with rate R_y in the future. The dispersion algorithm decides whether or not this peer should cache, and if so, which specific segments it should cache. The algorithm should ensure that, on the average, the same number of copies of each segment is cached, since all segments are equally important. To clarify, consider a file with only two segments. Keeping 90 copies of segment 1 and 10 copies of segment 2 means that we have effectively 10 copies of the media file available. In contrast, keeping 50 copies of each segment would result in 50 copies of the media file.

The `IndexDisperse` algorithm, shown in Figure 6, is to be run by the index server. Consider one media file with N segments, rate R Kb/s, and duration T hours. The algorithm requires the index server to maintain three types of information: per-peer information, per-cluster information, and per-system (or global) information. Table 1 summarizes the symbols used in the algorithm and their meaning.

For every peer P_x, the index server maintains: (1) N_x, the number of segments which are currently cached by P_x; (2) R_x, the rate at which P_x is willing to stream the cached segments; and (3) $u_x, 0 \leq u_x \leq 1$, the fraction of time P_x

Algorithm IndexDisperse

```
L_c ← 1, ∀c
while TRUE do
  Wait for a caching request
  /* Got request from peer P_y to cache N_y segments with rate R_y */
  c ← getCluster(P_y) /* identify client's cluster */
  Compute a_c, q_c, A, Q
  if q_c > a_c or Q ≫ (1/T)A then /* need to cache in round robin */
    if (L_c + N_y − 1) ≤ N then
      L_e = L_c + N_y − 1
    else
      L_e = N_y − (N − L_c + 1)
    end if
    Peer P_y caches from segment L_c to segment L_e
    L_c = L_e + 1
  end if
end while
```

Figure 6: Index-based dispersion algorithm.

is online. Recall that the peer is not available all the time.

For every cluster $\mathtt{c}$, the index server maintains the following: (1) $L_{\mathtt{c}}, 1 \leq L_{\mathtt{c}} \leq N$, the next segment to cache. (2) $q_{\mathtt{c}}$, the average request rate (per hour) the media file is being requested by clients from $\mathtt{c}$. $q_{\mathtt{c}}$ represents the required capacity in the cluster $\mathtt{c}$ per hour. (3) $a_{\mathtt{c}}$, the average number of copies of the movie cached by peers in cluster $\mathtt{c}$. $\mathtt{c}$ is computed from the following equation:

$$a_{\mathtt{c}} = \sum_{P_x \text{ in } \mathtt{c}} \frac{R_x}{R} \frac{N_x}{N} u_x. \quad (1)$$

The summation in Equation (1) computes the effective number of copies available in the cluster. It accounts for two facts: first, peers are not always online (through the term u_x), and second, peers do not cache all segments at the full rate (through the term $R_x N_x / RN$). Dividing $a_{\mathtt{c}}$ by T results in the number of requests that can be satisfied per hour, since every request takes T hours to stream. Hence, $(1/T)a_{\mathtt{c}}$ represents the available capacity in the cluster $\mathtt{c}$ per hour.

The index server maintains two global variables: (1) $A = \sum_{\mathtt{c}} a_{\mathtt{c}}$, the average number of copies of the movie cached by all peers in the system. (2) $Q = \sum_{\mathtt{c}} q_{\mathtt{c}}$, the average movie request rate in the system. Q and $(1/T)A$ represent the global required capacity and the global available capacity in the system, respectively.

The algorithm proceeds as follows. Upon getting a request from peer P_y to cache N_y segments, the index server identifies the cluster $\mathtt{c}$ of the requesting peer. Then, it computes $a_{\mathtt{c}}$, $q_{\mathtt{c}}$, A, and Q[1]. The algorithm decides whether P_y caches based on the available and the required capacities in the cluster. If the demand is larger than the available capacity in the cluster, P_y is allowed to cache N_y segments in a *cluster-wide round robin* fashion. To clarify, suppose we have a 10-segment file. $L_{\mathtt{c}}$ is initially set to 1. If peer P_1 sends a request to cache 4 segments, it will cache segments 1, 2, 3, and 4. $L_{\mathtt{c}}$, the next segment to cache, is now set to 5. Then, peer P_2 sends a request to cache 7 segments. P_2 will cache segments 5, 6, 7, 8, 9, 10, and 1. $L_{\mathtt{c}}$ is updated to 2, and so on. This ensures that we do not over cache some segments and ignore others.

Furthermore, the `IndexDisperse` algorithm accounts for the case in which some clusters receive low request rates while others receive very high request rates in a short period. In this case, the global required capacity Q is likely to be much higher than the global available capacity $(1/T)A$, i.e., $Q \gg (1/T)A$. Therefore, even if the intra-cluster capacity is sufficient to serve all requests within the cluster, the peer is allowed to cache if $Q \gg (1/T)A$ in order to reduce the global shortage in the capacity. The operator $\gg$ used in comparison is relative and can be tuned experimentally.

5 Evaluation

In this section, we evaluate the proposed P2P architecture through extensive simulation experiments. First, we study the performance of the P2P model under various situations, e.g., different client arrival patterns and different levels of cooperation offered by the peers. We are interested in the following performance measures as the system evolves over the time:

1. The overall system capacity, defined as the average number of clients that can be served *concurrently* per hour;

[1]Computing these quantities is not necessarily performed for every request, especially if the request arrival rate is high. Rather, they can be updated periodically to reduce the computational overhead. Also, these quantities are *smoothed* averages, not instantaneous values.

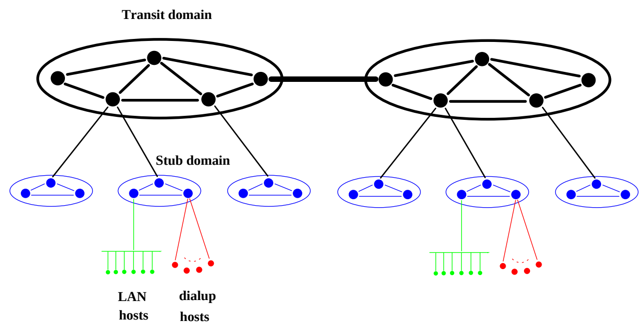


Figure 7: Part of the topology used in the simulation.

2. The average waiting time for a requesting peer before it starts playing back the media file;
3. The average number of satisfied (or rejected) requests; and
4. The load on the seeding server.

Second, we evaluate the proposed cluster-based dispersion algorithm and compare it against a random dispersion algorithm. The comparison criteria are: (1) the percentage of the requests satisfied within the same cluster, and (2) the reduction of the load on the underlying network due to careful dissemination of the media files over the participating peers.

We simulate a large (more than 13,000 nodes) hierarchical, Internet-like, topology. We use the GT-ITM tool [1] for generating the topology and the Network Simulator *ns-2* [19] in the simulation.

5.1 Performance of the P2P Architecture

5.1.1 Topology

Figure 7 shows a part of the topology used in the simulation. Approximately resembling the Internet, the topology has three levels. The highest level is composed of transit domains, which represent large Internet Service Providers (ISPs). Stub domains; which represent small ISPs, campus networks, moderate-size enterprise networks, and similar networks; are attached to the transit domains on the second level. Some links may exist among stub domains. At the lowest level, the end hosts (peers) are connected to the Internet through stub routers. The first two levels are generated using the GT-ITM tool [1]. We then, probabilistically add dialup and LAN hosts to routers in the stub domains.

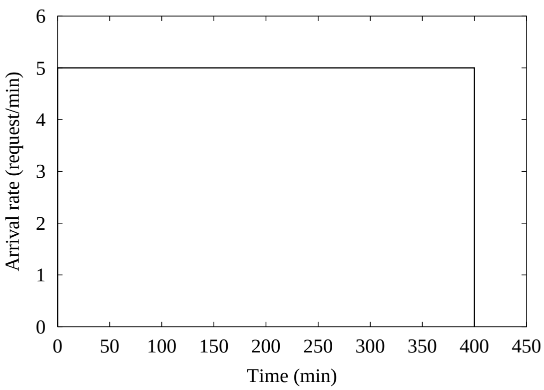
The topology used in this set of experiments consists of 20 transit domains, 200 stub domains, 2,100 routers, and a total of 11,052 hosts distributed uniformly at random. More details about the topology generation as well as the simulation scripts and programs are available at [20].

5.1.2 Simulation Scenario

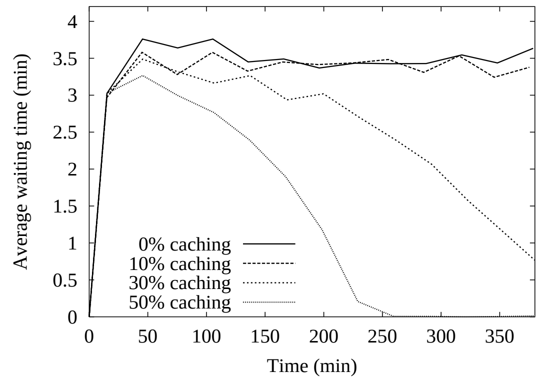
We simulate the following scenario. A seeding server with a limited capacity introduces a media file into the system. According to the simulated arrival pattern, a peer joins the system and requests the media file. Then, the `P2PStream` protocol, described in Section 3, is applied. We do not assess the overhead imposed by the searching step in this set of experiments. If the request can be satisfied, i.e., there is a sufficient capacity in the system, connections are established between the supplying peers and the requesting peer. Then, a streaming session begins. The connections are over UDP and carries CBR traffic. If the requesting peer does not find all the segments with the full rate, it backs off and tries again after an exponentially increased waiting time. If the waiting time reaches a specific threshold, the request is considered “rejected” and the peer does not try again. When the streaming session is over, the requesting peer caches some of the segments depending on the level of cooperation, called the caching percentage. For instance, if the caching percentage is 10% and the media file has 20 segments, the peer stores two randomly-chosen segments. The peer also selects a rate at which it wants to stream the cached segments to other peers.

5.1.3 Simulation Parameters

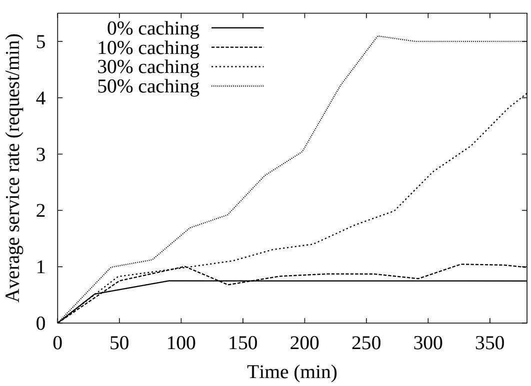
We have the following fixed parameters:



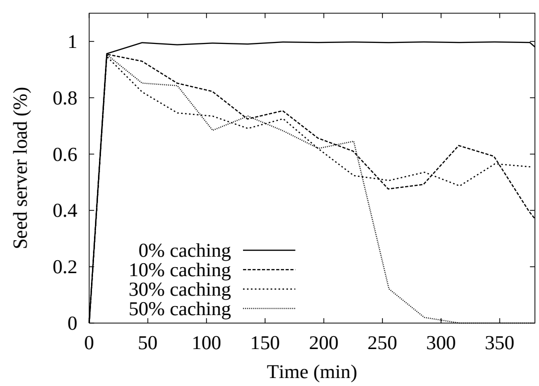
(a) Clients arrival pattern: constant rate arrivals



(b) Average waiting time



(c) Average service rate



(d) Load on the seeding server

Figure 8: Performance of the P2P architecture under constant rate arrivals.

1. A media file of 20 minute duration recorded at a CBR rate of 100 Kb/s and divided into 20 one-minute segments;
2. The dialup peers are connected to the network through links with 1 Mb/s bandwidth and 10 ms propagation delay;
3. The LAN peers have 10 Mb/s Ethernet connectivity with a 1 ms propagation delay;
4. The backbone links have a bandwidth of 155 Mb/s with variable delays, depending on whether a link is between two routers in the same stub domain, the same transit domain, or across domains;
5. The seeding server has a T1 link with a bandwidth of 1.5 Mb/s, which means that it can support up to 15 concurrent clients;
6. The requesting peer can open up to 4 connections with other peers to get a segment at the desired rate of 100 Kb/s; and
7. The maximum waiting time for a requesting client is two minutes.

We vary the caching percentage from 0% to 50% and study the system under various client arrival patterns. 0% caching means that the requesting peer does not store any segment of the media file; whereas with 50% caching, it stores half of the file. The results are summarized in the follow subsections.

5.1.4 Results for Constant Rate Arrivals

Figure 8 shows the behavior of the P2P architecture when the constant rate arrival pattern shown in Figure 8.a is applied to the system. Figure 8.c shows how the system's capacity evolves over the time. The average service rate, increases with the time, because as the time passes more peers join the system and contribute some of their resources to serve other requesting peers. The capacity is rapidly amplified, especially with high caching percentage. For instance, with 50% caching, the system is able to satisfy all the requests submitted at 5 requests/minute after about 250 minutes (about 4.2 hours) from the starting point. We can use Figure 8.c to answer the following two questions. Given a target client arrival rate, what should be the appropriate caching percentage? How long will it take for the system to reach the steady state (in which all clients are served)? To illustrate, suppose that the target service rate is 2 requests/minute. Then, 30% caching will be sufficient and the steady state will be achieved within less than 5 hours. The average waiting time, shown in Figure 8.b, is decreasing over the time, even though the system has more concurrent clients, as shown in Figure 8.d. This is due to the rapid amplification of the capacity.

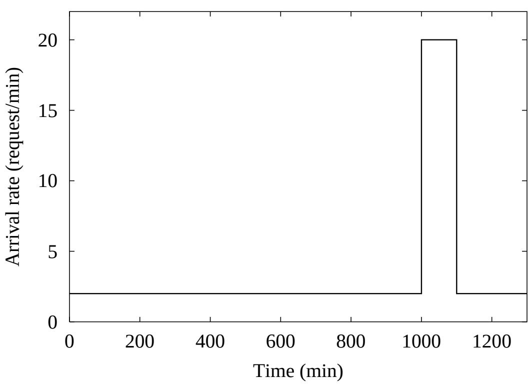
Finally, Figures 8.c and 8.d verify the diminishing role of the seeding server. Although the number of *simultaneous* clients increases until it reaches the maximum (limited by the arrival rate), the proportion of these clients that are served by the seeding server decreases over the time, especially with high caching percentages. For instance, with 50% caching and after about 5 hours, we have 100 concurrent clients, i.e., 6.7 times the original capacity, and none of them is served by the seeding server. Reducing the load on the seeding server is an important feature of the P2P streaming architecture, because it means that the seeding servers need not to be powerful machines with high network connectivity. Besides being moderate machines, the seeding servers are used only for a short period of time. Therefore, the cost of deploying and running these seeding servers (in case of a commercial service) is greatly reduced.

5.1.5 Results for Flash Crowd Arrivals

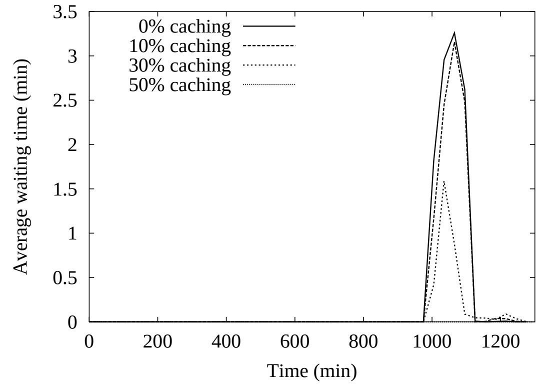
Flash crowd arrivals are characterized by a surge increase in the clients arrival rates. These kind of arrival patterns arise in cases such as releasing a popular movie or a recording of a publically interesting event. To simulate the flash crowd arrivals, we initially subject the system to a small request rate of 2 requests/minute for some period of time (*warm up* period), and then suddenly increase the arrival rate 10 folds to be 20 requests/minutes for a limited time (100 minutes). The arrival pattern is shown in Figure 9.a.

The results shown in Figure 9 demonstrate an appealing characteristic of the P2P architecture, namely the ability to handle flash crowd arrivals. For 50% caching, the average service rate in the system, shown in Figure 9.c, reaches as high as the clients arrival rate (i.e., 20 requests/min) during the crowd period. Therefore, the system does not turn away any customers, when the caching percentage is 50%. Moreover, all customers are served without having to wait, as shown in Figure 9.b.

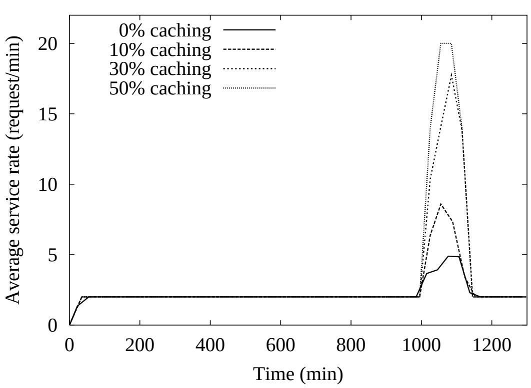
During the crowd period and with 50% caching, Figure 9.c indicates that there are as many as 400 concurrent clients in the system. This is an increase of 26.7 times in the original capacity. Even with that many clients, Figure 9.d shows that none of the clients is being served by the seeding server, which confirms that the seeding server's role is still just seeding the media file into the system. Finally, we notice that for caching percentages lower than 50%, the system needs a longer *warm up* period to cope with the flash crowd without the help of the seeding server.



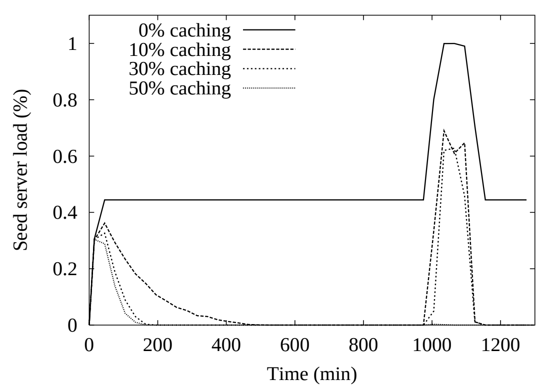
(a) Clients arrival pattern: flash crowd arrivals



(b) Average waiting time



(c) Average service rate



(d) Load on the seeding server

Figure 9: Performance of the P2P architecture under flash crowd arrivals.

5.1.6 Results for Poisson Arrivals

We subject the system to Poisson arrivals with different mean arrival rates. The results are not shown to conserve space, the reader is referred to [9]. The results are similar to the case of constant rate arrivals, except that there are more fluctuations due to the probabilistic nature of the Poisson arrivals. The results indicate the ability of the P2P architecture to handle statistically multiplexed client arrival patterns.

5.2 Evaluation of the Dispersion Algorithm

Since we are not aware of any existing dispersion algorithms that can be used in our model, we compare our cluster-based dispersion algorithms against a random dispersion algorithm. We evaluate the efficiency of the dispersion algorithm by measuring the average number of network hops traversed by the requested stream. Smaller number of network hops indicates savings in the backbone bandwidth and less susceptibility to congestion, since traffic passes through fewer routers.

5.2.1 Topology and Simulation Parameters

In this set of experiments, most of the parameters are the same as in the previous set of experiments, except that the topology is larger. The topology has 100 transit domains, 400 stub domains, 2,400 routers, and a total of 12,021 hosts. This topology is chosen to distribute the peers over a wider range, and hence stresses the dispersion algorithms more than the previous topology.

We vary the caching percentages from 5% to 90%. Low caching percentages, e.g., 5% and 10%, stress the dispersion algorithm more than the higher caching percentages. With low caching percentages, a peer stores few segments. Therefore, it is important for the dispersion algorithm to carefully choose these few segments. In contrast, with high caching percentages, a peer stores most of the segments, leaving little work for the dispersion algorithm. The clients arrive to the system according to a constant rate arrival pattern with a rate of 1 request/minute.

5.2.2 Simulation Scenario

The simulation scenario is similar to the scenario in the previous set of experiments with one difference in the last step of the `P2PStream` protocol (the caching step). For each caching percentage, we run the experiment twice. In the first run, we use a random dispersion algorithm, in which a peer *randomly* selects a specific number of segments (determined by the caching percentage) and store them locally. In the second run, we use the `IndexDisperse` algorithm, which caches the same number of segments but selects them carefully.

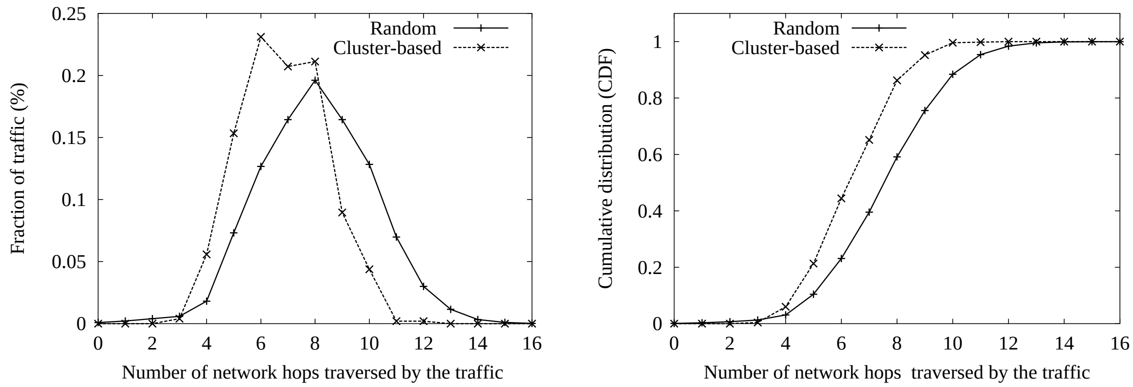
Each experiment lasts for 500 minutes of simulation time. For every streaming packet transmitted during the simulation, we measure the number of network hops that packet traverses. At the end of each experiment, we compute the distribution of the number of network hops traversed by all packets of the streaming traffic. We plot both the probability mass function (pmf) and the cumulative distribution function (CDF). The results are summarized in the following subsections.

5.2.3 Results for 5% Caching

Figure 10.a shows the pmf of the number of network hops for both the random and the `IndexDisperse` dispersion algorithms. The pmf curve of the `IndexDisperse` algorithm is shifted to the left of the random algorithm. This indicates that the traffic crosses fewer number of hops using the `IndexDisperse` algorithm than using the random algorithm. The arithmetic mean of the number of network hops for the random algorithm is 8.0520, while it is 6.8187 for the `IndexDisperse` algorithm. The saving is about 15.3% of the total bandwidth needed in the backbone. Given that a good streaming service requires a huge bandwidth, our dispersion algorithm achieves considerable savings.

The cumulative distribution, Figure 10.b, shows that about 44% of the traffic crosses six or less hops using our algorithm, whereas this value is only 23% for the random algorithm. A reasonable ISP network would have an average network diameter in the vicinity of six hops. This means that our dispersion algorithm keeps about 44% of the traffic within the same domain (cluster), which is often a desirable property for both the clients and the network.

Similar results were obtained for other caching percentages but not shown here to conserve space, we refer the reader to the technical report [9].



(a) Probability mass function (pmf) of the number of network hops

(b) Cumulative distribution function (CDF) of the number of network hops

Figure 10: Comparison between the random and the `IndexDisperse` dispersion algorithms, 5% caching.

6 Related Work

Significant research effort has addressed the problem of efficiently streaming multimedia, both live and on demand, over the best-effort Internet. Directly related to our work are systems like *SpreadIt* [3] for streaming live media and *CoopNet* [14], [13] for both live and on-demand streaming. Both systems build distribution trees using application-layer multicast and, like ours, they rely on cooperating peers. Multicast (network- or application-layer) is the basis for several other media delivery systems [4] [5]. Our work is different from these systems, since we do not use multicast in any form and our system is more appropriate for on-demand media service.

In the client/server world, proxies and caches are deployed at strategic locations in the Internet to reduce and balance load on servers and to achieve a better service. Content Delivery Network (CDN) companies such Akamai [15] and Digital Island [16] follow similar approaches to provide media streaming and other services. Our approach does not require any powerful proxies or caches. Rather, it uses peers' extra resources as numerous tiny caches. These tiny caches do not require large investment and collectively enlarge the capacity of the system in a way that potentially outperforms any powerful centralized caches.

The distributed video streaming framework [12] is also relevant to our work. The framework allows for multiple senders to feed a single receiver. The receiver uses a rate allocation algorithm to specify the sending rate for each sender to minimize the total packet loss. This specification is based on estimating the loss rate and the available bandwidth between the receiver and each of the senders. The authors assume that senders are capable of providing the rates computed by the rate allocation algorithm. In our case, the providing peers decide on the rates at which they are willing to provide.

Recently, the peer to peer paradigm has attracted the attention of numerous researchers. Two main categories of research can be identified: research on protocols and algorithms (mainly on searching), and research on building P2P systems. Location and routing protocols such as CAN [21], Chord [26], Pastry [22], and Tapestry [29] guarantee locating the requested object within a logarithmic number of steps, if the object exists in the system. However, they lack the flexibility of supporting keyword queries and in many cases (except for Pastry) they do not explicitly consider network locality. Other searching techniques do not provide such guarantees but they support flexible queries [28]. On the systems side, Gnutella [18] is the largest currently running file-sharing system. Freenet is another file-sharing system focusing on the *anonymity* of both the producer and consumer of the files [17]. Examples of large-scale storage systems built on top of P2P architectures are presented in [2], [11], and [23]. Our proposed system adds one more to the list but with a new service, namely, media streaming.

7 Conclusions and Future Work

We presented a P2P media streaming model that can serve many clients in a cost effective manner. We presented the details of the model and showed how it can be deployed over the current Internet. Specifically, we presented a P2P streaming protocol used by a participating peer to request a media file from the system; a cluster-based dispersion algorithm, which efficiently disseminates the media into the system; and a searching algorithm to locate nearby peers who have segments of the requested media file.

Through a large-scale simulation, we showed that our model can handle several types of client arrival patterns, including suddenly increased arrivals, i.e., flash crowds. Our simulation also showed that the proposed cluster-based dispersion algorithm reduces the load on the underlying network and keeps a large portion of the traffic within the same network domain.

We are currently embarking on implementing a prototype of the P2P media streaming system. The objective is to better assess to the proposed model and to demonstrate its applicability for a wide deployment. Addressing the security and robustness issues of the model are parts of our future work.

References

[1] K. Calvert, M. Doar, and E Zegura. Modeling internet topology. In *IEEE Communications Magazine*, pages 35:160–163, 1997.

[2] F. Dabek, M. Kaashoek, D. Karger, D. Morris, I. Stoica, and H. Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. In *Proc. of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII*, pages 71–76, Elmau/Oberbayern, Germany, May 2001.

[3] H. Deshpande, M. Bawa, and H. Garcia-Molina. Streaming live media over peer-to-peer network. Technical report, Stanford University, 2001.

[4] A. Dutta and H. Schulzrinne. A streaming architecture for next generation internet. In *Proc. of ICC'01*, Helsinki, Finland, June 2001.

[5] L. Gao and D. Towsley. Threshold-based multicast for continuous media delivery. *IEEE Transactions on Multimedia*, 3(4):pp. 405–414, December 2001.

[6] P. Golle, K. Leylton-Brown, and I. Mironov. Incentives for sharing in peer-to-peer networks. In *Proc. of The Second workshop on Electronic Commerce (WELCOM'01)*, Heidelberg, Germany, November 2001.

[7] M. Hefeeda and B. Bhargava. Cost-profi t analysis of a peer-to-peer media streaming architecture. Technical report, CERIAS, Purdue University, June 2002.

[8] M. Hefeeda and B. Bhargava. Peer-to-peer on-demand media streaming. Technical report, CERIAS TR 2002-xx, Purdue University, 2002. Work in progress.

[9] M. Hefeeda, B. Bhargava, and D. Yau. Pimss: Peer-to-peer internet media streaming service. Technical report, CERIAS TR 2002-20, Purdue University, June 2002.

[10] B. Krishnamurthy and J. Wang. On network-aware clustering of web clients. In *Proc. of ACM SIGCOMM*, Stockholm, Sweden, August 2000.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, Boston, MA, November 2000.

[12] T. Nguyen and A. Zakhor. Distributed video streaming over internet. In *Proc. of Multimedia Computing and Networking (MMCN02)*, San Jose, CA, USA, January 2002.

[13] V. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *Proc. of The 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, USA, March 2002.

[14] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. In *Proc. of NOSSDAV'02*, Miami Beach ,FL, USA, May 2002.

[15] Akamai Home Page. http://www.akamai.com.

[16] Digital Island Home Page. http://www.digitalisland.com.

[17] Freenet Home Page. http://www.freenet.sourceforge.com.

[18] Gnutella Home Page. http://www.gnutella.com.

[19] Napster Home Page. http://www.napster.com.

[20] PIMSS Home Page. http://www.cs.purdue.edu/homes/mhefeeda.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, San Diago, CA, USA, August 2001.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[23] A. Rowstron and P. Druschel. Storage management in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[24] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking (MMCN02)*, San Jose, CA, USA, January 2002.

[25] The Network Simulator. http://www.isi.edu/nsnam/ns/.

[26] I. Soitca, R. Morris, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, San Diago, CA, USA, August 2001.

[27] D. Xu, M. Hefeeda, S. Hambrusch, and B. Bhargava. On peer-to-peer media streaming. In *Proc. of IEEE ICDCS*, Vienna, Austria, July 2002.

[28] B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. In *Proc. of ICDCS'02*, Vienna, Austria, July 2002.

[29] B. Zaho, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.