A Cost-Effective Peer-to-Peer Architecture for Large-Scale On-Demand Media Streaming

Xin Liu
Department of Computer Science, University of British Columbia, Vancouver, Canada
Email: liu@cs.ubc.ca

Son T. Vuong
Department of Computer Science, University of British Columbia, Vancouver, Canada
Email: vuong@cs.ubc.ca

***Abstract*—This paper presents a cost-effective peer-to-peer (P2P) architecture for large-scale on-demand media streaming, named BitVampire. BitVampire's primary design goal is to aggregate peers' storage and upstream bandwidth to facilitate on-demand media streaming. To achieve this goal, BitVampire splits published videos into segments and distributes them to different peers. When a peer (or a receiver) wants to watch a video, it (i) searches the corresponding segments, then (ii) selfishly determines the best subset of supplying peers and (iii) aggregates bandwidth from these peers to stream the media content. In BitVampire, participating peers help each other to get the desired media content, thus powerful servers/proxies are not necessary, which makes it a cost-effective approach. To demonstrate the effectiveness of BitVampire, we conducted extensive simulation on large, hierarchical, Internet-like topologies. We also implemented a functional prototype using Java and Java Media Framework (JMF) to demonstrate the feasibility of BitVampire.**

***Index Terms*—Peer-to-Peer, On-Demand Media Streaming**

I. INTRODUCTION

On-demand media streaming has recently gained intensive consideration due to its promising usage in a rich set of Internet-based services such as video on demand, distance learning, media distribution, etc. However, there are still many challenges toward building efficient, scalable on-demand streaming systems due to the high bandwidth and delay requirements for media streaming.

A majority of existing on-demand media streaming systems follows the Client-Server design, in which videos are stored in a set of central servers. All the requests are served by servers and the streaming content is directly delivered from servers. Obviously, this architecture is not scalable since the servers will become bottleneck as the requests increase. To save servers' resources and alleviate servers' traffic load, several proxy-based architectures have been proposed, in which a set of proxies are deployed in the network. Clients can request the cached portion of videos from the proxies.

However, in both server-based solutions and proxy-based solutions, the servers and proxies are expected to deliver high-quality streaming service to a large number of clients. Therefore, the servers and proxies should be very powerful in terms of computing power, upstream bandwidth, storage, etc., which makes the deployment and maintenance cost very expensive. On the other hand, recent research and experiments reveal that the current Internet has enough resources to support large-scale media streaming in a peer-to-peer manner [18][23]. If these resources can be aggregated in a systematic way, an on-demand media streaming system could be constructed without powerful servers/proxies, which makes the system cost-effective. This is the primary design goal of BitVampire.

To achieve this goal, BitVampire splits the published videos into segments and distributes them to different peers. When a peer (or a receiver) wants to watch a video, it first searches the corresponding segments, then selfishly determines the best subset of supplying peers and aggregates bandwidth from these peers to stream the video. To deploy this architecture in a dynamic heterogeneous peer-to-peer network, three problems need to be addressed: (1) How to distribute and cache segments, taking into consideration that peers offer different resources and may leave at any time. (2) How to efficiently find the desired segments. (3) How to aggregate bandwidth from multiple peers and coordinate them to serve one streaming request. In this paper, we present our approaches to address these three problems, with the emphasis on the first and the third one.

The basic idea of our architecture, as well as some preliminary experiments and initial prototype implementation were presented in the previous paper [13]. However, this paper is substantially different in that: (1) a new scheduling algorithm for multiple-source streaming is proposed. Compared to the previous one, our new

Based on "A Peer-to-Peer Framework for Cost-Effective On-Demand Media Streaming", by Xin Liu, Jun Wang, and Son T. Vuong which appeared in the Proceedings of the IEEE Consumer Communications and Networking Conference 2006, Las Vegas, USA, January 2006. © 2006 IEEE.

© 2006 ACADEMY PUBLISHER

scheduling algorithm is simpler but more efficient. (2) extensive simulation-based performance evaluation is presented in this paper.

The rest of the paper is organized as follows. Section II gives an overview of the system, identifying the system entities and operations. Section III briefly introduces Category Overlay [12], which is chosen as the underlying search infrastructure. Section IV presents the system design, including the segments distributing and caching scheme, segments searching scheme, and the scheduling algorithm that coordinates multiple peers to deliver media content to receiver in the real-time mode. The simulation setup and performance evaluation are presented in Section V. Section VI presents the prototype implementation. We present related work in Section VII, and conclude this paper in Section VIII.

II. System Overview

This section provides a system overview of the proposed architecture. We first identify the system entities. Then, we explain how the system works.

A. System Entities

Following are the entities in our proposed architecture:

- *Peers*. This is a set of computers participating in the system. Each participating peer contributes some of its upstream bandwidth and storage. We denote the upstream bandwidth and storage peer P_i is willing to contribute as Bw_i and St_i, and the available upstream bandwidth and storage peer P_i can contribute at a specific time as Bw^{avail}_i and St^{avail}_i. Initially, $Bw^{avail}_i = Bw_i$, $St^{avail}_i = St_i$ and $Bw^{avail}_i \leq Bw_i$, $St^{avail}_i \leq St_i$ hold at any time.
- *Seed Peers*. To handle the situation in which all the hosting peers of a specific segment leave the system, we introduce seed peers into the system. Seed peers always stay in the system, and each segment of published videos has a replica stored in seed peers. Seed peers serve the streaming request only when the request cannot be satisfied by regular peers, and if it is the case, seed peers will re-distribute a replica of that segment to peers, thus decreasing the future demand on seed peers. Seed peers are almost the same as regular peers, except that they are stable and have large storage, which is very cheap nowadays.
- *Media Files*. This is a set of videos published in the system. Every video is assigned a unique ID, called *videoID*, which is generated when the video is published. Every video belongs to a predefined type, such as Action video, Sports video, Comedy video, etc., and is associated with a list of keywords provided by the publisher. We assume that each video is encoded with a constant bit rate *Br* (in kbps). A video is split into equal-sized segments, and segment is the minimum unit that a peer can cache.

B. System Operations

In BitVampire, when a peer publishes a video, the video will be split into equal-sized segments, and these segments will be distributed to peers according to our segments distributing algorithm. Once peers receive segments, they will publish the received segments to the Category Overlay. Note that during the segments distributing process, every segment will have a replica distributed to one of the seed peers.

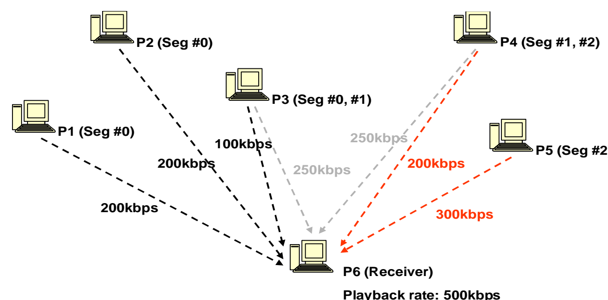


Figure 1. Example of watching a video

When a peer (or a receiver) wants to watch a video, it first searches the 1st segment. Then it determines if the streaming request can be satisfied by the peers contained in the search results (including seed peers). If the answer is yes, it selfishly determines the best subset of supplying peers and applies the proposed scheduling algorithm to aggregate bandwidth from the selected supplying peers and coordinate them to stream the 1st segment; otherwise, the request is rejected. When the streaming of the 1st segment is almost over, the receiver will do the same thing with the 2nd segment, the 3rd segment, and so on. Fig. 1 shows an example. Suppose peer P_6 wants to watch a video whose playback bit rate is 500kbps. It searches for segment #0 and finds that P_1, P_2, P_3 have segment #0; it then selects P_1, P_2, P_3 as the supplying peers and aggregates bandwidth from them to stream segment #0. Segment #1 and #2 are streamed in the same way.

After the streaming session of a segment is over, the receiver will cache the segment in its contributed storage. We use Least Recently Used (LRU) algorithm to select the victim segment to replace if there is not enough available storage for the new cached segment.

III. Category Overlay

In this section, we briefly introduce Category Overlay, which is chosen as the underlying search infrastructure in BitVampire.

The basic idea of Category Overlay is to construct multiple category specific overlays on the unstructured P2P system and restrict a specific search within the corresponding overlay. In more detail, we first cluster the peer group into clusters. Then in each cluster, nodes (called *Agent Nodes*) are selected to take charge of predefined categories. The *Agent Node* is responsible for maintaining a keyword list table (called *Content Index Table*) for all the contents belonging to the categories it is in charge of. For a specific category, all of its *Agent Nodes* (in different clusters) are connected to form a category overlay. Thus, multiple category overlays can be

© 2006 ACADEMY PUBLISHER

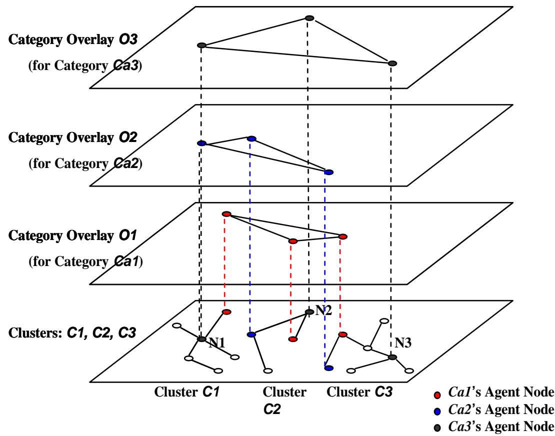


Figure 2. Example of Category Overlay

constructed over the clusters.

Fig. 2 shows an example of Category Overlay. As the figure shows, peers are clustered into three clusters: C_1, C_2, and C_3. In each cluster, nodes are selected to take charge of three predefined categories: Ca_1, Ca_2, and Ca_3. For example, in cluster C_1, node N_1 is in charge of category Ca_3; in cluster C_2, node N_2 is in charge of category Ca_3; and in cluster C_3, node N_3 is in charge of category Ca_3. Since nodes N_1, N_2, and N_3 are *Agent Nodes* for category Ca_3, they are connected to form the category overlay O_3. Category overlay O_1 and O_2 can be formed in the same way.

In Category Overlay, every cluster member node maintains a *Category Table*, which stores the Category-to-Agent mappings. It looks like a hash table where the key is a category and the value is that category's *Agent Node*. When a node publishes a content belonging to a specific category, it first looks up its *Category Table* to find that category's *Agent Node*, then it sends a "publish content" message to the found *Agent Node*, along with the keyword list (while the content still reside in the owner's storage). Upon receiving this message, the *Agent Node* will store the keyword list in its *Content Index Table*.

When a node issues a query, it specifies a category, and provides a list of keywords. The query will go to the *Agent Node* which is in charge of that specified category. Then the corresponding *Agent Node* looks up its *Content Index Table* to find the contents that have the matched keywords, and returns the results to the query initiator. In addition, that *Agent Node* also needs to propagate the query within the corresponding overlay. Each *Agent Node* in this overlay will look up its *Content Index Table* and return the results to the query initiator. Compared to Gnutella [6], in which queries need to go through all the nodes, a query in Category Overlay just needs to be propagated within the corresponding overlay, which is much more efficient.

Note that in Category Overlay, each cluster is tree-based (the root of the tree is called *Core Node*). Two cluster members are connected with *Cluster Links* (tree branches), and two neighbour clusters are connected through *Inter-Cluster Links*. More information about Category Overlay and its maintenance mechanisms, as well as the simulation-based performance evaluation can be found in [12].

IV. SYSTEM DESIGN

In this section, we present the system design of BitVampire. We first discuss our segments distributing algorithm, then describe the segments publishing, searching, and caching schemes. Finally, we discuss our scheduling algorithm that coordinates multiple peers to stream the segment, and compare it with other two possible solutions.

A. Segments Distributing

In BitVampire, each participating peer P_i estimates its stay time in the system by computing the smoothed weighted average as follows and uses this value to represent its stability.

$$EstimatedStay_i = \alpha \times EstimatedStay_i + \beta \times CurrentStay_i \cdot \quad (1)$$

where $EstimatedStay_i$ is the estimated stay time of peer P_i, taking into account all the stay history of P_i, and $CurrentStay_i$ is the time period P_i participated in the system since its last leave or failure. $\alpha + \beta = 1$, α is between 0.8 and 0.9, and β is between 0.1 and 0.2. Besides, peer P_i also maintains the average usage ratio of its contributed bandwidth since it participated in the system, called $R^{usage}{}_i$, and the frequency it serves streaming requests in the recent period, called $Freq^{serve}{}_i$.

We define $G^{St}{}_i$, the goodness of candidate peer P_i to store a segment as a function of its $EstimatedStay_i$, Bw_i, $R^{usage}{}_i$, and $Freq^{serve}{}_i$. Suppose there are m candidate peers: $\{P_1, P_2, \ldots, P_m\}$, $G^{St}{}_i$ has the following form:

$$G^{St}{}_i = \alpha_{St} \times \frac{EstimatedStay_i}{\max\limits_{1 \le i \le m}\{EstimatedStay_i\}} + \beta_{St} \times \frac{Bw_i \times (1 - R_i^{usage})}{\max\limits_{1 \le i \le m}\{Bw_i \times (1 - R_i^{usage})\}} - \gamma_{St} \times \frac{Freq^{serve}{}_i}{\max\limits_{1 \le i \le m}\{Freq^{serve}{}_i\}} \quad (2)$$

where α_{St}, β_{St}, γ_{St} are the factors to give $EstimatedStay_i$, $Bw_i \times (1\text{-}R^{usage}{}_i)$, $Freq^{serve}{}_i$ different weights. In our simulation and prototype implementation, α_{St} is set to 0.35, β_{St} is set to 0.25, and γ_{St} is set to 0.4. Given this equation, the candidate peer that is more stable, has higher available bandwidth and lower serve frequency will have a greater G_{St}.

To detect node failure, every peer in the Category Overlay periodically sends "alive" messages to its parent. We let peer send its *EstimatedStay*, *Bw*, R^{usage}, and $Freq^{serve}$ along with the "alive" message. The parent node collects information contained in the received "alive" messages and periodically sends an aggregate report to its parent, along with the "alive" message. Thus, eventually *Core Node* will have recent information of every cluster member. *Core Node* sorts the cluster members by their G^{St} in descending order and stores the result in a sorted candidates list. *Core Node* periodically maintains this list based on the renewed information of the cluster members.

When a peer wants to publish a video, it splits the video into equal-sized segments. The workload analysis

© 2006 ACADEMY PUBLISHER

of today's enterprise media server [2] found that most of clients only watch the first several minutes portion of media files. To benefit from this fact, we let the first segment have several replicas. Suppose the video is split into N_s segments, and the first segment has N_f replicas, then totally N_s+N_f segments need to be distributed to peers.

The publishing peer sends a "publish video" message to its cluster's *Core Node*, and this message will be propagated to the *Core Nodes* of other clusters. After receiving this message, the *Core Node* selects the first N_c ($N_c>N_s+N_f$) peers from the sorted candidates list and sends the information of these peers back to the publisher. The publisher waits for $Timeout_p$ to receive the messages sent back by the *Core Nodes* and collects information of the candidate peers. After $Timeout_p$, the publisher will assign segments to candidate peers.

Fig. 3 illustrates the pseudo code of our Media Segments Distributing (MSD) algorithm. Note that the algorithm tends to assign more segments to the candidate peers which have higher G^{St}, which means these peers will take more responsibilities to serve streaming requests. However, their $Freq^{serve}$ will increase as the streaming requests come in, thus decreasing their G^{St}. When another video is published, it is likely that their G^{St} will be exceeded by others, so that video's segments will be distributed to other peers. In a long term, this could result in load balance in peers to some extent.

```
/* in this algorithm, we do not differentiate between original segment
   and its replica, they are referred same as segment. */
Input:
  candidateList:  candidate peers sorted by G^St in descending order;
  num_candidates: number of candidates;
  num_segs:       number of segments;
  num_replicas:   number of replicas for the first segment;
Assigning:
  j = 1;
  for (i = 0; i < num_segs + num_replicas; i++) {
     select j^th node in candidateList, suppose the selected node is N_k;
     if (i < num_replicas + 1)
        assign segment 0 to node N_k;
     else
        assign segment (i - num_replicas) to node N_k;
     if (j == num_candidates)
        j = 1;
     else
        j++;
  }
```

Figure 3. Media segments distributing (MSD) algorithm

Once the segments assignment is done, the publisher will send segments to peers. When a peer receives a segment, it checks if there is enough storage available (St^{avail} >= *seg_size*, where *seg_size* is the size of a segment). If yes, it stores the received segment and decreases its St^{avail} by *seg_size*; otherwise, it uses Least Recently Used (LRU) algorithm to select a victim segment to replace.

B. Segments Publishing and Searching

After segments are distributed to peers, peers will publish received segments to Category Overlay. As mentioned in Section III, to publish content in Category Overlay, a node should specify the category the content belongs to, as well as a keywords list. In our proposed architecture, we define the categories as follows: (1) We first predefine video types, such as Action video, Sports video, Comedy video, etc.; (2) Then we combine video type and segment number as the category, such as Action-0, Action-1, Sports-0, etc. So all the 1st segments of Action videos belong to Action-0 category; all the 2nd segments of Action videos belong to Action-1 category; and so on.

Note that when a peer publishes a video, it should specify the video type, and provide a list of keywords. When the publisher distributes segments to peers, the specified video type and the keywords list will be sent to peers as well. When a peer publishes a received segment, it will use the combination of video type and segment number as the segment's category, and use the received keywords list as the publishing keywords. In addition, each published segment has a *videoID* to specify which video it comes from (as mentioned in Section II, every video has a unique *videoID*). Thus when searching, we can use this *videoID* to ensure that the found segments come from the same video. After segments have been published to Category Overlay, we can search the desired segments in the same way described in Section III.

C. Segments Caching and Seed Re-Distributing Mechanism

Once a peer finishes watching a segment, it will cache this segment in its contributed storage. We use this cache policy because we believe that the possibility of peer re-watching this segment is relatively higher than others. However, the storage contributed by a peer is limited, so we adopt Least Recently Used (LRU) as the cache replacement algorithm, in which the least recently used segment will be chosen as the victim if there is not enough available storage for the new cached segment.

Another issue needs to be addressed is that at some point, it is possible that a specific segment is missing, because all of its host peers leave the system or fail. To handle this situation, we deploy several seed peers in the network. Seed peers always stay in the system and every segment of published videos has a replica stored in one of them. Seed peers serve the streaming request only when the request cannot be satisfied by regular peers. To alleviate seed peers' load, we propose a Seed Re-Distributing (SRD) mechanism, in which when the seed peer offers help to stream a segment, it will distribute a replica of that segment to peers, thus decreasing the future demand on seed peers. Seed peers do not need to be powerful machines with high upstream bandwidth. They just need to be stable and have enough storage capacity.

D. Segments Streaming

1) Supplying Peers Selection: When a peer (receiver) searches the desired segments for watching, the size of the results could be large. Thus we need a scheme to select supplying peers from the search results. We let the receiver selfishly determine the best subset of supplying peers. The details of our scheme are presented below.

© 2006 ACADEMY PUBLISHER

After receiving the search results, the receiver will send an enquiry message to each peer contained in the results. Upon receiving this message, a peer will send a reply message back to the receiver, along with its Bw^{avail} and *EstimatedRTT*, where *EstimatedRTT* is the estimated round trip time between the peer and the receiver. The receiver waits for $Timeout_e$ to get the reply messages and collects information contained in the messages. After $Timeout_e$, the receiver will select the subset of supplying peers based on their G^{Sp}, the goodness of peer to become supplier. Suppose there are m candidate peers: $\{P_1, P_2, \ldots, P_m\}$, the $G^{Sp}{}_i$ for a peer P_i is defined as follows:

$$G^{Sp}{}_i = \alpha_{Sp} \times \frac{Bw^{avail}{}_i}{\max\limits_{1 \le i \le m}\{Bw^{avail}{}_i\}} - \beta_{Sp} \times \frac{EstimatedRTT_i}{\max\limits_{1 \le i \le m}\{EstimatedRTT_i\}}. \quad (3)$$

where α_{Sp}, β_{Sp} are the factors to give $Bw^{avail}{}_i$, $EstimatedRTT_i$ different weights. In our simulation and prototype implementation, α_{Sp} is set to 0.4, and β_{Sp} is set to 0.6. Given this equation, the candidate peer that is nearer to the receiver, has higher available bandwidth will have a greater G^{Sp}.

The receiver will select M (in our simulation and prototype implementation, M = 3) candidate peers that have the greatest G^{Sp} as the suppliers, as long as the aggregated available bandwidth from these peers is bigger than or equal to the video playback bit rate. Otherwise, more than M peers will be selected to meet the playback bit rate requirement. The unselected candidate peers will be kept in a *standby* set, from which substitute peers can be selected in case of suppliers leave or failure. If the aggregated available bandwidth from all of the candidate peers is less than the video playback bit rate, the segment watching request will be rejected.

After supplying peers have been selected, the receiver will reserve bandwidth from them. Suppose M supplying peers ($P_1, P_2, \ldots, P_M$) are selected, and the video playback bit rate is Br. The receiver will reserve bandwidth $Bw^r{}_i$ from supplier P_i in proportion to its $G^{Sp}{}_i$, and satisfy the following conditions:

- The reserved bandwidth is in multiple of bandwidth reservation unit (in our simulation and prototype implementation, the unit is set to 64kbps).
- $\sum_{i=1}^{M} Bw^r{}_i = Br$

Then the receiver sends a "reserve bandwidth" message to each supplier. Upon receiving this message, supplying peer P_i decreases its $Bw^{avail}{}_i$ by $Bw^r{}_i$. When the streaming session supplied by peer P_i is over, P_i increases its $Bw^{avail}{}_i$ by $Bw^r{}_i$. Note that by "reserve bandwidth", we do not mean that the bandwidth $Bw^r{}_i$ from peer P_i is actually reserved and cannot used by other applications. The current Internet does not provide resource reservation service, thus the bandwidth contributed by supplying peer P_i may fluctuate during the streaming session.

2) Multiple-Source Scheduling Algorithm: To fully utilize the aggregate bandwidth from multiple supplying peers, we want different suppliers to send different portion of a segment to the receiver at the same time. Thus we further divide each segment into equal-sized blocks, so the receiver can parallel download different blocks from different supplying peers in the real-time model. In our simulation and prototype implementation, one block contains one second video content.

During the streaming session, a supplier may leave or fail at any time. If that happens, the receiver will select another peer from the *standby* set to substitute the leaving/failing supplier, which is called *supplier switching*. During the *supplier switching* period, the aggregate bandwidth is less than the required playback bit rate, thus the receiver may experience buffer underflow. To cope with this, we require the receiver to buffer at least $S_{initBuff}$ blocks before the video playback starts. This is called *initial buffering*. The time to finish downloading $S_{initBuff}$ blocks is called *initial buffering time*.

Suppose a segment contains N blocks $\{blk_0, blk_1, \ldots, blk_{N-1}\}$ and the receiver selects M supplying peers $\{P_1, P_2, \ldots, P_M\}$, the scheduling problem can be stated as follows: given the contributed bandwidth from the supplying peers $\{Bw_1, Bw_2, \ldots, Bw_M\}$, where the sum of the contributed bandwidth equals to Br (the video playback bit rate), how to assign blocks to those supplying peers to achieve a minimum *initial buffering time*, as well as downloading the earlier block as early as possible.

There are two possible solutions. One is Round Robin (RR), where blocks are assigned to suppliers in a round robin manner. However, RR treats each supplier equally, no matter how much bandwidth it contributes to the streaming session. Thus some bandwidth contributed from the powerful peers could be wasted, while the weak peers may take too many blocks to send. To cope with this, another possible solution is assigning blocks to suppliers in proportion to their contributed bandwidth, which we call Bandwidth Proportional (BP). In this approach, supplier P_i sends $(Bw_i/Br)*N$ blocks, starting from whatever P_{i-1} ends. This approach fully uses the bandwidth from each supplier when sending blocks. However, for the blocks at the very beginning, only one supplier contributes its bandwidth, which inevitably results in a long *initial buffering time*. Taking all of these into consideration, a good schedule should be the one in which blocks are assigned to suppliers in a roughly round robin manner, but in each round, the blocks are assigned in proportion to the bandwidth contributed by suppliers.

Based on the discussion above, we propose a Multiple-Source Scheduling (MSS) algorithm, which assigns blocks to different suppliers to send. The algorithm is executed by the receiver to generate the schedule. Fig. 4 illustrates the pseudo code of the algorithm. The suppliers are sorted by their contributed bandwidth Bw in descending order. For a supplier *supplier[i]*, *time[i]* indicates its earliest start time to send current block. Initially, *time[i]* is set to 0. We assign blocks to suppliers starting from the first block. To assign current block *block[curr_blk]*, we first iterate all the suppliers to calculate the estimated finish time for each supplier to send current block and store the result in *estimatedTime[i]*. Then we find the minimum *estimatedTime* across all the suppliers. Suppose

© 2006 ACADEMY PUBLISHER

```
Input:
    num_suppliers :   number of suppliers;
    supplier[i] :     the suppliers sorted by Bw in descending order;
    Bw[i] :           contributed bandwidth by supplier[i];
    num_blks :        number of blocks;
    block[i] :        blocks;
    blk_size :        block size;

Scheduling:
    for (i = 1; i ≤ num_suppliers; i++)
        time[i] = 0;
    curr_blk = 0;
    while (curr_blk ≤ num_blks - 1) {
        for (i = 1; i ≤ num_suppliers; i++)
            estimatedTime[i] = time[i] + blk_size / Bw[i];
        select k, where estimatedTime[k] = Min {estimatedTime[i]};
        assign block[curr_blk] to supplier[k];
        time[k] = time[k] + blk_size / Bw[k];
        curr_blk ++;
        if (curr_blk > num_blks - 1)
            break;
    }
```

Figure 4. Multiple-source scheduling (MSS) algorithm

supplier[k] has the minimum *estimatedTime*, then current block *block[curr_blk]* is assigned to *supplier[k]* and *time[k]* is increased by *blk_size*/Bw_k (the time for *supplier[k]* to finish sending current block). At this point, *block[curr_blk]* is assigned. We repeat the same procedure to assign *block[curr_blk+1]*, *block[curr_blk+2]*, and so on, until we finish assigning the last block.

Unlike Round Robin (RR) and Bandwidth Proportional (BP) approaches, our Multiple-Source Scheduling (MSS) algorithm assigns blocks to suppliers based on their estimated finish times to send current block. The supplier that has the minimum estimated finish time will take the current block. This approach ensures that blocks are assigned to suppliers in proportion to their contributed bandwidth, and each block is downloaded by the receiver as early as possible after the previous blocks are received.

Fig 5. illustrates an example of assigning 8 blocks to 3 suppliers using RR, BP, and MSS respectively. Suppose *Br*, the video playback bit rate is 512kbps, and one block contains one second of the video content. There are 3 suppliers contributing their bandwidth, in which P_1 contributes 320kbps (5/8·*Br*), P_2 contributes 128kbps (1/4·*Br*), and P_3 contributes 64kbps (1/8·*Br*). As the figure shows, to download all the 8 blocks, RR takes 16 seconds, while BP and MSS takes only 8 seconds. If $S_{initBuff}$ (the number of initial buffering blocks) is set to 4, RR takes 8 seconds to download all of the 4 initial buffering blocks, BP takes 6.4 seconds, while MSS takes only 4.8 seconds. This example demonstrates that compared to RR and BP, MSS uses less time to download all the blocks, and achieves a small *initial buffering time*. Our simulation also verified this result.

3) Streaming Session: Once the schedule is generated, the receiver will send it to the suppliers. When a supplying peer receives the schedule, it will send the assigned blocks to the receiver in order using UDP and

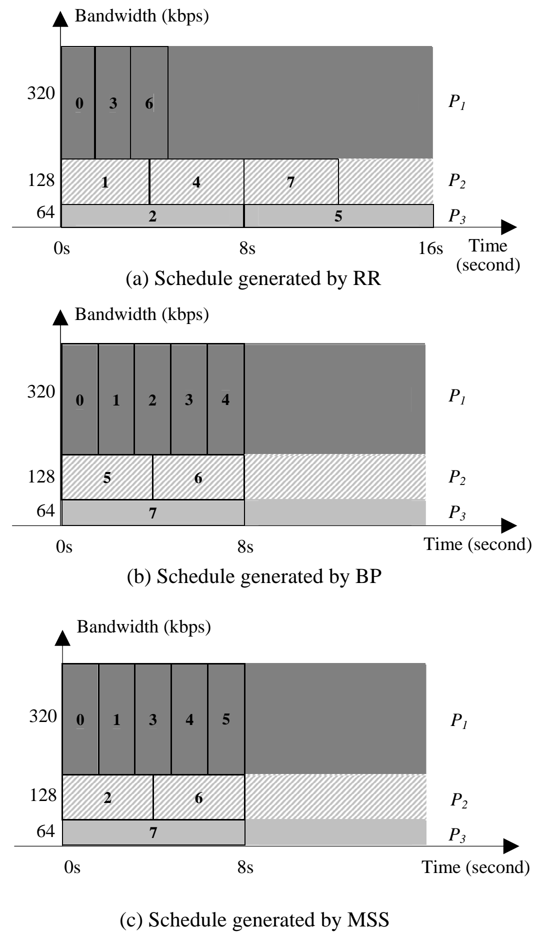


Figure 5. Assigning 8 blocks to 3 suppliers using RR, BP, MSS

perform TCP-friendly congestion control over the UDP connection (e.g., RAP [16] or TFRC [15]). As mentioned before, during the streaming session, some of the suppliers may leave or fail at any time, and the incoming streaming rates from the suppliers may decrease due to network congestion. In these cases, the *supplier switching* will happen, in which the receiver selects substitute supplying peers to replace the leaving/failing supplier or the supplier whose sending rate is decreased. After that, the receiver will generate a new schedule to assign the rest not-received blocks to the new set of suppliers. Besides, to cope with the UDP packets lost, the receiver will keep tracking of received blocks and make sure that every block is received T_{adv} seconds before the playback. Otherwise, the block is identified as lost, and the receiver will ask the corresponding supplier to re-send it.

In the client side, the receiver maintains a ring buffer. Once the receiver receives a block, it will write this block to the corresponding position of the ring buffer. After all the initial buffering blocks are received, the receiver will continuously read data from the ring buffer and render video frames on the player window.

V. SIMULATION

In this section, we evaluate the performance of our proposed architecture through extensive simulation

© 2006 ACADEMY PUBLISHER

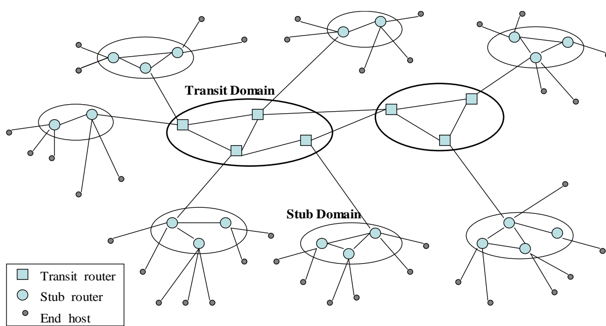


Figure 6. Part of simulation topology

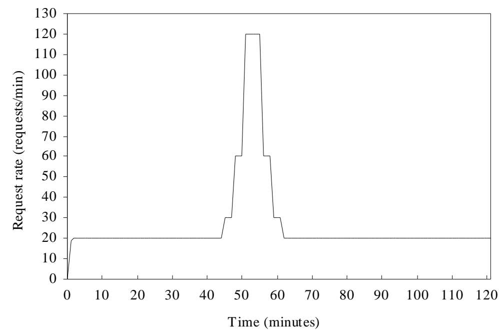


Figure 7. Flash crowd arrival pattern

experiments. We first describe the simulation setup, and then present the results.

A. Simulation Setup

1) Simulation Topologies: In all of the simulations, we use large hierarchical, Internet-like topologies. All of the topologies have three levels. The top level consists of several transit domains, which represent large Internet Service Providers (ISPs). The middle level contains several stub domains, which represent small ISPs, campus networks, moderate-sized enterprise networks, etc. (Each stub domain is connected to one of the transit domains). At the bottom level, end hosts (peers) are connected to stub domains. Fig. 6 shows a part of the topology used in the simulation. The first two levels (router-level) contain transit routers and stub routers, which are generated using GT-ITM tool [22]. We then randomly attach end hosts (peers) to stub routers with uniformed probability. Each experiment was run on 10 different topologies, and the results presented in this paper are the average results of experiments running in these 10 topologies. On the average, the topologies used in the simulations consist of 10 transit domains, 200 stub domains, 2050 routers, and a total of 3010 end hosts (peers), in which 6 hosts are selected as seed peers.

2) Simulation Parameters: Unless otherwise specified, all of our simulations use the following parameter settings.

During the simulation, there are totally 500 videos published in the network, each with 512kbps constant playback bit rate (CBR) and 1 hour length. Each video is split into 12 segments. The length of each segment is 5 minutes and the size is about 19MB. We let the first segment have 2 replicas ($N_f = 2$), and by default, receiver selects 3 ($M = 3$) supplying peers, if these peers have enough available bandwidth.

We assign bandwidths and delays to network links as follows: (1) Each link between two routers has bandwidth ranging from 6Mbps to 20Mbps, and delay ranging from 5ms to 40ms. (2) Each link between end hosts (peers) to routers has bandwidth ranging from 512kbps to 2Mbps, and delay ranging from 4ms to 10ms. The contributed upstream bandwidth and storage from peers are configured as follows: (1) Each peer contributes upstream bandwidth ranging from 128kbps to 1Mbps, and storage ranging from 2 segments (38MB) to 5 segments (95MB). (2) Each seed peer contributes upstream bandwidth ranging from 1Mbps to 2Mbps, and storage ranging from 1000 segments (19GB) to 3000 segments (57GB). Note that the configuration of peers in the experiments represents typical equipment settings for the current desktop PCs connected to the Internet. From the simulation results, we can see that based on these usual, low-cost PCs, our proposed architecture can support large-scale on-demand media streaming services.

To reflect the dynamic nature of peer-to-peer networks, we let 20 peers leave the system per minute. Each leaving peer will stay off-line for a period ranging from 15 minutes to 3 hours, and then rejoin the system. We evaluate the performance of the system under 2 different video requests arrival patterns: (1) constant arrival pattern, where every 3 seconds, a peer initiates a video watching request (request rate: 20 requests/min). (2) flash crowd arrival pattern (showed in Fig. 7), where at the beginning, peers request videos at the rate of 20 requests/min, and then suddenly increase to the rate of 120 requests/min (from the 44th minute to the 61st minute).

The popularity of videos follows Zipf-like distribution, where the popularity of the ith most popular video is proportional to $1/i^{\alpha}$. The authors of [2] reported that in a long period (in months), the video file access frequencies in HP corporate media solutions server and HPLabs media server follow Zipf-like distribution, with α ranging from 1.4 to 1.6. Therefore, in our simulation, we set α to 1.4.

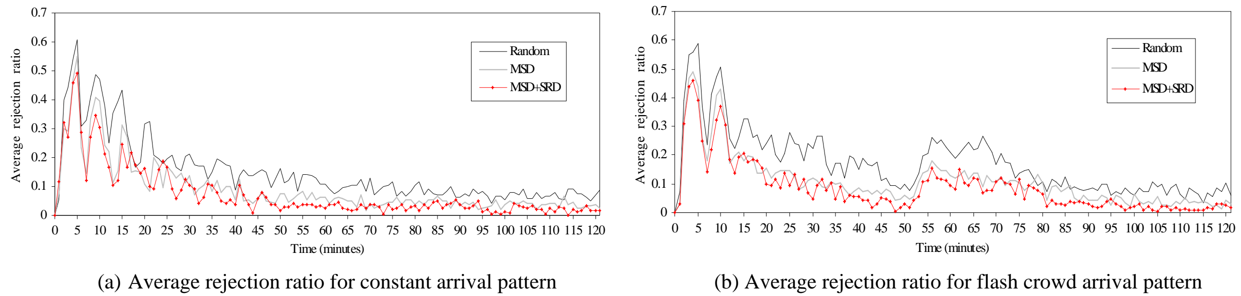
The last parameter setting is the length of video watching session. Again, based on the reports from [2], we let 50% of the sessions last 5~10 minutes (watching 1~2 segments), 30% of the sessions last 10~30 minutes (watching 2~6 segments), and 20% of the sessions last 30~60 minutes (watching 6~12 segments).

B. Simulation Results

We first evaluate the effectiveness of our media segments distributing (MSD) algorithm, and then we present the simulation results of our multiple-source scheduling (MSS) algorithm, which results in a small *initial buffering time*. Finally, we evaluate the performance our architecture in different sized peer-to-peer networks and under different peers' cooperation levels.

1) System Streaming Capacity Amplification: In this set of experiments, we show that our media segments distributing (MSD) algorithm plus seed re-distributing

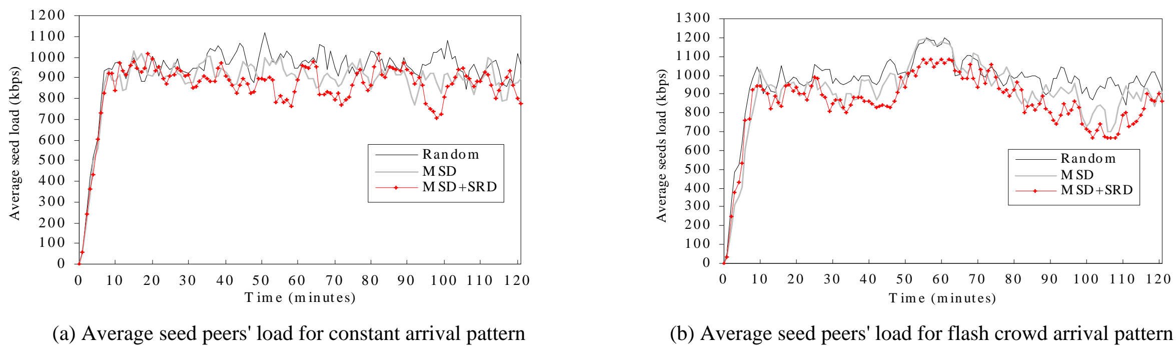
© 2006 ACADEMY PUBLISHER



(a) Average rejection ratio for constant arrival pattern

(b) Average rejection ratio for flash crowd arrival pattern

Figure 8. Average rejection ratio



(a) Average seed peers' load for constant arrival pattern

(b) Average seed peers' load for flash crowd arrival pattern

Figure 9. Average seed peers' load

(SRD) mechanism can result in a fast system streaming capacity amplification. We define the system streaming capacity as the number of video watching sessions that can be served concurrently, and use the simple random segments distributing algorithm as the comparison base. In our simulation, each video watching session may have different length; in another word, each session may contain different number of segment requests. Thus, we use segment requests rejection ratio as our measurement metric. A lower segment requests rejection ratio means that more segment requests can be accepted at a specific time, which results in a higher system streaming capacity.

Fig. 8(a) shows the simulation result for the constant video requests arrival pattern, and Fig. 8(b) shows the result for the flash crowd arrival pattern. We run the simulations for 2 hours, and compute the average segment requests rejection ratio every minute. From both of these two figures, we can see that compared to the random segments distributing algorithm, our MSD algorithm quickly decreases the rejection ratio, and if the SRD mechanism is applied, the rejection ratio is decreasing faster.

2) Seed Peers' Load: The next set of experiments evaluate the load on seed peers. As the previous experiments, we run the simulations for 2 hours, and compute the average load on seed peers every minute. Fig. 9(a) shows the simulation result for the constant video requests arrival pattern, and Fig. 9(b) shows the result for the flash crowd arrival pattern. As showed in both of the figures, the average seed peers' load with MSD algorithm is less than the load with random segments distributing algorithm. And if the SRD mechanism is applied, the seed peers' load will decrease further. The reason for this result is that MSD tries to distribute segments to the peers that are more stable, have more available bandwidth, thus decreasing the demand for seed peers. If SRD is applied, the segments that cannot served by the regular peers will be distributed to peers by seeds. Thus next time, requests for these segments can be served by the regular peers, therefore further reducing the seed peers' load.

Reducing load on seed peers is an important feature of the proposed architecture; because it means that the seed peers need not to be powerful machines with high upstream bandwidth. They just need to be stable and have large storage, which is very cheap nowadays.

3) Initial Buffing Time: This set of experiments evaluate the multiple-source scheduling (MSS) algorithm. We compare MSS algorithm with Round Robin (RR), showing that MSS can achieve smaller *initial buffering time*. We generate 5,000 video requests, in both constant arrival pattern and flash crowd pattern. We then record the *initial buffering time* for each accepted request using RR and MSS respectively. In our simulation, the initial buffering length is set to 8 blocks ($S_{initBuff} = 8$). Fig. 10 shows the simulation result. It is clear that MSS always

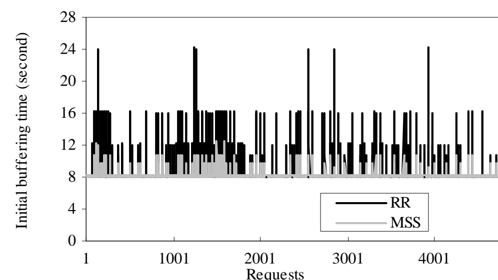


Figure 10. Initial buffering time using RR, MSS

© 2006 ACADEMY PUBLISHER

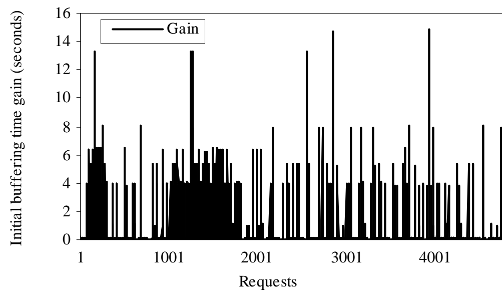


Figure11. Initial buffering time gain

achieves equal or smaller *initial buffering time* compared to RR. Fig. 11 shows the *initial buffering time* gain, where gain is defined as the *initial buffering time* using RR minus *initial buffering time* using MSS. As the figure shows, the gain is always bigger than or equal to 0, and can be as large as 14 seconds.

4) Varying Network Size: In this set of experiments, we evaluate the performance of our proposed architecture in different sized networks. We measure the average segment requests rejection ratio for 3 different sized P2P networks: (1) 3000 peers network (consists of 2050 routers and 3000 peers), (2) 6000 peers network (consists of 2050 routers and 6000 peers), and (3) 9000 peers network (consists of 2050 routers and 9000 peers). The video requests arrive in flash crowd pattern.

Fig. 12 shows the simulation results. From the figure, we can see that since the network size increases, more segment requests will be issued by peers at the same time, thus at the beginning, the rejection ratio for 6000 peers network is bigger than the one for 3000 peers network and the rejection ratio for 9000 peers network is bigger than the one for 6000 peers network. However, the rejection ratio decreases fast. After about 25 minutes, the rejection ratio for 6000 peers network is almost same as the one for 3000 peers network. And after about 35 minutes, the rejection ratio for 9000 peers network is almost same as the one for 3000 peers network. The simulation results verified our hypothesis that as more peers participating in the system, more segment requests can be supported at the same time, since more resources are contributed to the system by peers. The simulation results also imply that our proposed architecture is scalable, as long as the participating peers contribute some of their resources to the system.

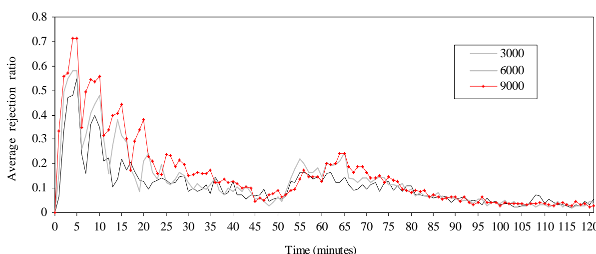


Figure 12. Average rejection ratio for various sized network

5) Varying Peers' Cooperation Level: Our final set of experiments evaluates system performance under different peers' cooperation levels. We measure the average segment requests rejection ratio for 3 different peers' cooperation levels: (1) Low cooperation level, where peers contribute their bandwidth ranging from 64kbps to 512kbps and storage ranging from 1 segment (19MB) to 3 segments (57MB); (2) Medium cooperation level, where peers contribute their bandwidth ranging from 128kbps to 1Mbps and storage ranging from 3 segments (57MB) to 6 segments (114MB); (3) High cooperation level, where peers contribute their bandwidth ranging from 1Mbps to 2Mbps and storage ranging from 5 segment (95MB) to 10 segments (190MB). We run the simulations on the 3010 peers network (consists of 2050 routers and 3010 peers). The video requests arrive in flash crowd pattern.

Fig. 13 shows the simulation results. From the figure, we can see that as the peers' cooperation level increases, the segment requests rejection ratio decreases faster, which means that the system streaming capacity is amplified faster. The reason for this is that if peers contribute more resources to the system, there will be more storage to cache segments and more bandwidth to support streaming requests; thus, the system streaming capacity increases faster.

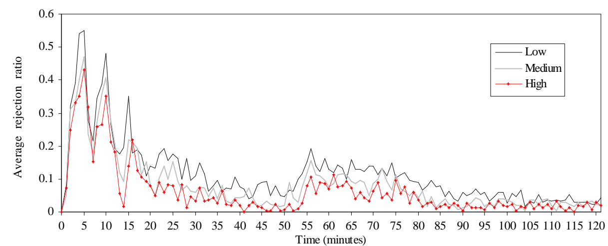


Figure 13. Average rejection ratio for various peers' cooperation level

VI. PROTOTYPE

To demonstrate the feasibility of BitVampire, we implemented a complex functional prototype using Java and Java Media Framework (JMF) [10]. In this section, we first present the prototype system architecture, then we briefly describe the prototype implementation.

A. Prototype System Architecture

In our prototype implementation, we choose Category Overlay [12] as the underlying search infrastructure. Thus, a Category Overlay prototype has been implemented as one part of BitVampire.

Fig. 14 shows the system architecture of our prototype, which consists of four layers: Core Layer, Service Layer, Abstraction Layer, and Application Layer. The key point of this layered design is to decouple of application logics, services, and underlying peers network from each other. For example, Abstraction Layer separate Application Layer from Service Layer. Thus, service algorithms (at Service Layer) can be modified or replaced without

© 2006 ACADEMY PUBLISHER

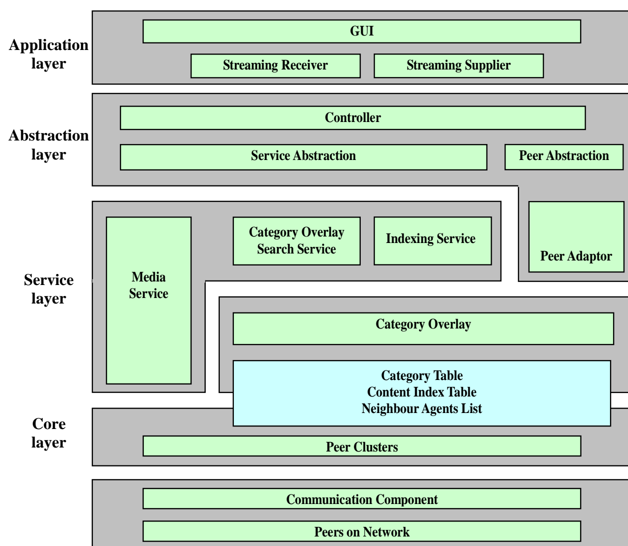


Figure 14. Prototype system architecture

changing or with only a little bit changing of the application logic code (at Application Layer).

B. *Prototype Implementation*

In the prototype implementation, control packets are sent using TCP and streaming packets are sent using UDP. To ensure TCP-friendly congestion control over UDP connection, RAP protocol [16] was used to adjust UDP packets sending rate. Fig. 15 is the snapshot of the prototype, where three nodes are running and two of them are watching videos.

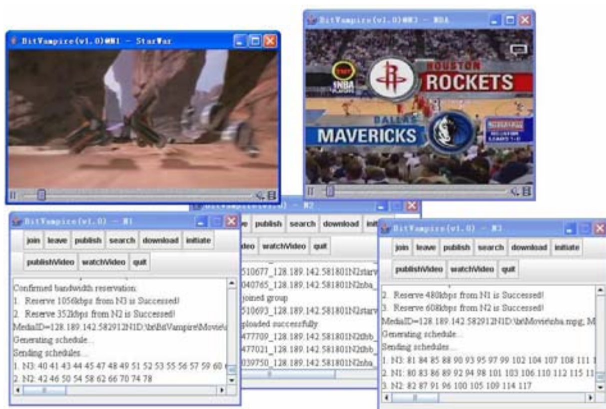


Figure 15. Snapshot of prototype

We also conducted some preliminary experiments in a local network environment, which consists of 10 desktop PCs residing in different labs of our department. The experiment results show that based on these common low-cost PCs, BitVampire can achieve smooth video playbacks in the receiver side.

VII. RELTATED WORK

This section presents the previous work related to our proposed architecture. We start from the conventional central server-based on-demand media streaming systems. Then we describe several proxy-based systems. Finally, we present an existing P2P-based system, and discuss its difference between our proposed architecture.

A. *Central Server-based Systems*

A majority of the existing on-demand media streaming systems follows Client-Server model, in which a set of centralized servers store all of the video files and respond to all of clients' requests. However, this architecture is not scalable since servers will become the bottleneck as the requests increase. To save servers' resources and alleviate servers' traffic loads, multicast has been applied and different solutions have been proposed. Batching [21] aggregates multiple client requests into one multicast session. However, the users have to suffer long playback delay since their requests are enforced to be synchronized. Patching [4][8] tries to address this problem by allowing the client to catch up with an on-going multicast session and patch the missing starting portion through server unicast. In merging [5], a client can repeatedly merge into a larger and larger multicast session using the same way as patching. However, in order to ensure smooth playback, these two approaches need the client to be capable of receiving multiple streams simultaneously and buffering large amount of data. In periodic broadcasting [9][20], the server separates a media stream into segments and periodically broadcasts them through different multicast channels, from which a client can choose to join in. Although more efficient in saving server's bandwidth, it shares the same limitations as the approaches mentioned above.

B. *Proxy-based Systems*

Another major category of on-demand media streaming systems employs the cooperative proxy caching technique. Existing work in this area includes prefix-based caching [17][19] and segment-based caching [1][3]. In prefix-based caching, proxies store the initial frames of popular clips. Upon receiving a request for the stream, the proxy initiates transmission to the client and simultaneously requests the remaining frames from the server. As the proxy is generally closer to the clients than the origin server, the start-up delay for a playback can be remarkably reduced. In segment-based caching, parts of media content are cached on different proxies in the network and the stream is coordinated to playback from these independent caches.

C. *P2P-based Systems*

Mohamed M. Hefeeda, et al. proposed a P2P on-demand media streaming architecture in their work [7]. This is probably the system most like ours so far. However, our architecture is substantially different from theirs in the following ways: (1) They cluster peers into two-level clusters, and super peers are selected from cluster members. Then they rely on these super peers to search the media content. In our architecture, we rely on Category Overlay to search the desired media segments, which not only provides an efficient keyword search service, also avoids the possible single point of failure

© 2006 ACADEMY PUBLISHER

problem due to introducing super peers. (2) In their architecture, a seed peer introduces a media file into the system. Initially the seed peer holds all of the segments. As the streaming requests come in, the accessed segments will be cached in requesters. However, in our architecture, once the video is published, it will be split into segments and these segments will be distributed to different peers. The segments distributing process improves the service availability, since the published video is belonging to the whole network, not the single publishing peer. (3) In their work, they do not consider the user behaviour when watching video. Their architecture requires that all the segments of a video be found before the streaming starts. However, this is not a reasonable restriction because most streaming sessions last only for a short period. Our architecture removes this restriction, where users can watch any segments if there is enough available resources currently in the network. (4) To aggregate bandwidth from multiple supplying peers, their architecture simply assigns a different portion of the segment to peers in proportion to their bandwidth. In our architecture, a more complicated schedule algorithm is proposed to coordinate multiple supplying peers to stream the segment. (5) In their simulation study, the system contains only one video. While in our simulation experiments, the system has 500 videos published, and we also take into consideration the videos' popularity and the user behaviour pattern when watching the video.

VIII. Conclusion and Future Work

In this paper, we propose BitVampire, a novel cost-effective peer-to-peer architecture for large-scale on-demand media streaming. In this architecture, published videos are split into segments and distributed to different peers, thus upstream bandwidth form multiple peers can be aggregated to serve a single video streaming request. We propose a media segments distributing (MSD) algorithm to distribute the segments to peers and rely on Category Overlay, an application-level overlay network, to efficiently search the desired media segments. However, BitVampire can sit on top of any search infrastructure as long as it provides efficient content searching services. To parallel download streaming content from multiple sources (supplying peers) in the real-time mode, we further divide segments into blocks and propose a multiple-source scheduling (MSS) algorithm to assign blocks to different supplying peers to send.

We conducted extensive simulation experiments on large, hierarchical, Internet-like topologies. The simulation results show that the proposed architecture can support large-scale on-demand media streaming services in a dynamic heterogeneous peer-to-peer network. The results also demonstrate that the proposed media segments distributing (MSD) algorithm can achieve a fast system streaming capacity amplification, and the proposed multiple-source scheduling (MSS) algorithm can achieve a small *initial buffering time*. We also implemented a complex functional prototype using Java and Java Media Framework (JMF) and conducted some preliminary experiments in a local network environment, which demonstrates the viability of BitVampire.

In the near future, we plan to deploy our prototype on the PlanetLab [14] to test its performance in the real Internet environment. We also plan to adopt some adaptive streaming technologies to improve the quality of service. One possible approach is to use layered coding, in which a video is encoded into multiple layers. The more layers a peer receives, the better video quality it will get. The receiver decides how many layers it can receive based on the current bandwidth from the supplying peers. If network congestion happens, the receiver can ask supplying peers to send fewer layers, which results in a smooth quality adaptation. Another possible approach is to use the priority drop technique [11], in which the supplying peers drop some less important packets if they detect network congestion.

References

[1] S. Acharya and B. Smith, "MiddleMan: A Video Caching Proxy Server", in *Proc.* of *NOSSDAV '00*, Chapel Hill, NC, 2000.

[2] L. Cherkasova, M. Gupta, "Charactering Locality, Evolution, and Life Span of Accesses in Enterprise Media Server Workloads," in *Proc. of NOSSDAV'02*, Miami Beach, FL, 2002.

[3] Y. Chae, K. Guo, M. Buddhikot, S. Suri and E. Zegura, "Silo, Tokens, and Rain-bow: Schemes for Fault Tolerant Stream Caching", *Special Issue of IEEE JSAC on Internet Proxy Services*, 2002.

[4] Y. Cai, K. Hua and K. Vu, "Optimized Patching Performance", in *Proc. of ACM/SPIE Multimedia Computing and Networking (MMCN '99)*, San Jose, CA, January 1999.

[5] D. Eager, M. Vernon and J. Zahorjan, "Minimizing Bandwidth Requirements for On-Demand Data Delivery", *IEEE Transactions on Knowledge and Data Engineering* 13(5), 2001.

[6] Gnutella. http://www.gnutella.com

[7] M. M. Hefeeda, B. K. Bhargava, D. K. Y. Yau, "A Hybrid Architecture for Cost-Effective On-Demand Media Streaming", *Computer Networks 44 (2004)*.

[8] K.A. Hua, Y. Cai and S. Sheu, "Patching: A Multicast Technique for True On-Demand Services", in *Proc. of ACM Multimedia'98*, Bristol, UK, 1998.

[9] K.A. Hua and S. Sheu, "Skyscraper Broadcasting: A new Broadcasting Scheme for Metropolitan VOD systems", in *Proc. of ACM SIGCOMM '97*, Cannes, French Riviera, France, 1997.

[10] Java Media Framework. http://java.sun.com/products/java-media/jmf/

[11] C. Krasic, J. Walpole, W. C. Feng, "Quality-Adaptive Media Streaming by Priority Drop", in *Proc. of NOSSDAV'03*, Monterey, CA, June 2003.

[12] X. Liu, J. Wang and S. T. Vuong, "A Category Overlay Infrastructure for Peer-to-Peer Content Search," *in Proc. of APDCM'0*5 (in conjunction with *IPDPS'05*), Denver, CO, April 2005.

[13] X. Liu, J. Wang, and S. T. Vuong. A Peer-to-Peer Framework for Cost-Effective On-Demand Media Streaming. In *Proc. of CCNC'06*, Las Vegas, NV, January 2006.

[14] PlanetLab. http://www.planet-lab.org/

© 2006 ACADEMY PUBLISHER

[15] Padhye, J. Kurose, D. Towsley, and R. Koodli, "A model-based TCP-friendly rate control protocol", in *Proc. of NOSSDAV'99*, Basking Ridge, NJ, 1999.
[16] R. Rejaie, M. Handley, and D. Estrin, "RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet", in *Proc. of IEEE INFOCOM'99*, New York City, NY, 1999.
[17] S. Ramesh, I. Rhee and K. Guo, "Multicast with Cache (mCache): An Adaptive Zero-delay Video-on-Demand Service", in *Proc. of INFOCOM '01*, Anchorage, AK, 2001.
[18] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang, "The Feasibility of Supporting Large-Scale Live Streaming Applications with Dynamic Application End-Points", *in Proc. of SIGCOMM'04*, Portland, OR, 2004.
[19] S. Sen, J. Rexford, and D. Towsley, "Proxy Prefix Caching for Multimedia Streams", in *Proc. of IEEE INFOCOM'99*, New York City, NY, 1999.
[20] S. Viswanathan and T. Imielinski, "Metropolitan Area Video-on-Demand Service using Pyramid Broadcasting", *Multimedia Systems 4*, 1996.
[21] G. O. Young, C.C. Aggarwal, J.L. Wolf and P.S. Yu, "On Optimal Batching Policies for Video-on-Demand Storage Servers", in *Proc. of International Conference on Multimedia Computing and Systems* (*ICMCS'96)*, 1996.
[22] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to Model an Internetwork", in *Proc. of IEEE INFOCOM'96*, April 1996.
[23] X. Zhang, J. Liu, B. Li, and T. P. Yum, "CoolStreaming/DONet: A Data-driven Overlay Network for Peer-to-Peer Live Media Streaming", in *Proc. of IEEE INFOCOM'05*, Miami, FL, 2005.

Xin Liu received the B.Eng. degree from Tsinghua University, Beijing, China, in 1999, the M.Sc. degree from Peking University, Beijing, China, in 2002, and the M.Sc. degree from University of British Columbia, Vancouver, BC, Canada, in 2005. He is currently working toward the Ph.D. degree in computer science from University of British Columbia, Vancouver, BC, Canada.

From 2001 to 2002, he was interned at Microsoft Research Asia, Bejing, China for about 9 months.

His research interests lie in computer networking, multimedia networking, distributed systems, and peer-to-peer networks, with an emphasis on building systems and services that can be deployed on the Internet.

Son T. Vuong received the B.Sc. degree from California State University, CA, USA, in 1972, the M.Eng. degree from Carleton University, Ottawa, ON, Canada, in 1977, and the Ph.D. degree in computer science from University of Waterloo, Waterloo, ON, Canada, in 1982.

He is currently an associate professor at the Computer Science Department, University of British Columbia, Vancouver, BC, Canada. His research interests are in protocol engineering, multimedia and wireless communications, high speed networks, mobile intelligent agents, distributed systems and architecture.

© 2006 ACADEMY PUBLISHER